The VLDB Journal (2022) 31:897–925
https://doi.org/10.1007/s00778-021-00701-5

SPECIAL ISSUE PAPER

General graph generators: experiments, analyses, and improvements

Sheng Xiang[1,2] · Dong Wen[2] · Dawei Cheng[3] · Ying Zhang[2] · Lu Qin[2] · Zhengping Qian[4] · Xuemin Lin[5]

Received: 21 February 2021 / Revised: 9 July 2021 / Accepted: 7 September 2021 / Published online: 7 October 2021

Abstract
Graph simulation is one of the most fundamental problems in graph processing and analytics. It can help users to generate new graphs on different scales to mimic observed real-life graphs in many applications such as social networks, biology networks, and information technology. In this paper, we focus on one of the most important types of graph generators: general graph generators, which aim to reproduce the properties of the observed graphs regardless of the domains. Though a variety of graph generators have been proposed in the literature, there are still several important research gaps in this area. In this paper, we first give an overview of the existing general graph generators, including recently emerged deep learning-based approaches. We classify them into four categories: simple model-based generators, complex model-based generators, autoencoder-based generators, and GAN-based generators. Then we conduct a comprehensive experimental evaluation of 20 representative graph generators based on 17 evaluation metrics and 12 real-life graphs. We provide a general roadmap of recommendations for how to select general graph generators under different settings. Furthermore, we propose a new method that can achieve a good trade-off between simulation quality and efficiency. To help researchers and practitioners apply general graph generators in their applications or make a comprehensive evaluation of their proposed general graph generators, we also implement an end-to-end platform that is publicly available.

Keywords Graph generator · Graph neural networks · Graph simulation · Experimental evaluation

1 Introduction

Due to the graph's strong expressive power, a host of researchers in fields such as e-commerce, cybersecurity, social networks, military, public health, and many more, are turning to graph modeling to support real-world data analysis [11,22,67–70,78]. For instance, the graph can be used to model the interactions between compounds and proteins in bioinformatics for drug discovery where each node represents the compound or a protein, and the interactions between them are captured by the edges [28,29,43,59,75]. In a social network, a node can represent a user and an edge can represent the relationship (e.g., friendship) between two users [7,50,53].

In the graph processing and analytics, a key step is the collection or generation of the graph data. In some applications, it is important to use graph generators (i.e., graph-generative models) to generate simulated graphs based on the real-life graph data for two reasons: (1) the inaccessibility of the whole real-life graphs; and (2) a better understanding of the distribution of graph structures and other features. For instance, as highlighted in [61], data acquisition is key step in responsible

✉ Dawei Cheng
dcheng@tongji.edu.cn

Sheng Xiang
xiangsheng218@gmail.com

Dong Wen
dong.wen@uts.edu.au

Ying Zhang
ying.zhang@uts.edu.au

Lu Qin
lu.qin@uts.edu.au

Zhengping Qian
zhengping.qzp@alibaba-inc.com

Xuemin Lin
lxue@cse.unsw.edu.au

1 Zhejiang Gongshang University, Hangzhou, China

2 AAII, University of Technology Sydney, Sydney, Australia

3 Tongji University, Shanghai, China

4 Alibaba Group, Hangzhou, China

5 University of New South Wales, Sydney, Australia

data management, and it is essential to collect or generate *representative* data. In some scenarios, users are only able to obtain a small sample of the real-life graph due to various limits such as incomplete observability, privacy concern, and company/government policy. It is necessary to use representative graphs with similar size and distribution to the real-life graphs for the training or performance evaluation purpose. For instance, it is a common practice to start the system development and data collection at the same time, especially when the data collection is cost-consuming (e.g., in the counter-terrorist applications) or the two tasks are managed by two separate teams. To better tune or validate the efficiency and scalability of the algorithms during system development, it is desirable to use representative large-scale simulated graphs before the real large-scale graph is readily available. Another example is the collaboration between the graph computing team at Alibaba Group and the finance data-analysis team at Ant Group for graph pattern-based fraud detection. Specifically, the graph computing team aims to develop efficient and scalable graph pattern-detection algorithms to find abnormal graph patterns in the finance network such that the finance data-analysis team can quickly identify some potential threats. As the finance network data are sensitive and cannot be directly released, the graph simulator has been deployed to generate a large-scale finance network for the graph computing team. Moreover, graph generators can provide a large number of simulated graphs for the training of the graph-based learning models [29,75]. By learning the distribution of the real-life graphs, the graph generators can also help to better understand the real-life graphs. For instance, graph generators can be used to generate source code [12] and formulas [76], which help to understand insights of the graph data. Graph generators can be used to obtain node representations of large networks [34] and to extract multiple relation semantics from knowledge graphs [71]. Molecular graph generators extract the distribution of compounds and design new and reasonable drugs [60,75]. Some researchers use graph generators to create neural network structures for the model architecture search [72,74].

1.1 Motivation

Due to the graph generators' importance in directly related applications, there is a long history of the study of graph generators in many domains such as database, data mining and, machine learning. Readers can refer to a recent survey [11] for a comprehensive overview of this line of research. In this paper, we focus on *general graph generators* which aim to reproduce structural properties of observed graphs regardless of the domains, something that is fundamental in the study of graph-generative models. Despite the existence of many outstanding achievements, we note that there still exist several unsolved important problems. These form the underlying motivation of this experimental paper.

1. **No comprehensive overview on emerging deep learning-based general graph generators** With the advance of deep learning techniques, advanced generative models such as autoencoder and generative adversarial network (GAN) have been widely used for data generation in many fields (e.g., image and audio), significantly enhancing the performance of traditional approaches. Not surprisingly, these techniques have also been applied recently to graph generators (e.g., [10,18,42,73,77]). However, to our best knowledge, there is no comprehensive overview of these emerging deep generative graph models in the literature. For instance, in their recent survey paper [11] on graph generators, the authors of [11] only briefly mention several machine learning-based general graph generators in the sections devoted to challenges and open problems.
2. **No systematic and comprehensive performance comparison of general graph generators** By their nature, graphs are complex, making it difficult to capture explicitly the distribution of observed graphs. For instance, a graph with n nodes can be represented by up to $n!$ equivalent adjacency matrices, each corresponding to a different, arbitrary node ordering/numbering. Moreover, we need to learn distributions over possible graph structures without assuming a fixed set of nodes (e.g., to generate candidate molecules of varying sizes). This being the case, traditional graph similarity metrics (e.g., graph edit distance [56] and maximum common subgraph [44]) cannot be applied to determine whether two graphs are from the same distribution. Therefore, unlike other data distributions where the (dis)similarity of two sets of objects (e.g., point sets in Euclidean space) can be measured directly by a numeric value (e.g., KL divergence [36] and Earth Mover Distance [14]), we have to resort to the distributions of various graphs' properties (e.g., degree distribution); that is, if two graphs are from the same distribution, the corresponding distribution of particular properties (e.g., degree distribution) should be very similar. This creates a large number of metrics for evaluating the likelihood of two graphs from various perspectives, such as Maximum Mean Discrepancy [77] (MMD) between two node-degree distributions. In addition to the value of graph simulation, there are also many evaluation metrics for general graph generators that are critical for decision-making by researchers and practitioners under different application scenarios, such as training time, inference time, scalability, permutation invariance, and tuning difficulty. To the best of our knowledge, existing studies usually consider only a few

metrics in their performance evaluations, and many worthy of attention are overlooked. Moreover, the number of competitors and graphs deployed in experiments is rather limited. We also note that there are discrepancies in experimental results reported in certain papers. For instance, it is reported in [42] that the simulation quality of the GRAN [42] outperformed the GraphRNN-S [77] on the protein dataset, while our experiments have different observations.

3. **No algorithm that can achieve a good trade-off between graph simulating quality and efficiency (scalability)** Traditional graph generators usually rely on well-defined graph models, and their corresponding graph simulation algorithms are typically efficient and scale well to large graphs. However, these techniques are hand-engineered to model a particular family of graphs and lack the capacity to learn the generative model directly from observed real-life graphs. For instance, the B-A model [3] is carefully designed to capture the scale-free nature of empirical degree distributions, but fails to capture many other aspects of real-world graphs such as community structures. On the contrary, the emerging deep generative models can achieve far superior graph simulating quality, but suffer from low efficiency and scalability when the deep learning techniques are applied to general graph generators. For instance, the graph generators based on RNN (e.g., GraphRNN [77]) need to store long node ordering to infer the adjacency matrix of the whole graph, which consumes considerable time and space. Although some research efforts aim to enhance the efficiency of deep neural network-based approaches, the results are not very promising in terms of the trade-off between simulation quality and efficiency (scalability). For instance, GRAN [42] accelerates graph simulation by generating a block of nodes per step, but it still needs to infer the whole graph, whose adjacency matrix requires $O(n^2)$. Due to the limit of floating-point operations per second (FLOPS) and GPU's memory, it cannot generate a graph with more than 10^5 nodes in our experimental environment.
4. **No handy toolkit for the users of general graph generators** To increase the impact of a specific type of technique, support from handy software libraries or toolkits is critical in enabling users to apply these techniques easily to their applications. One well-known example is the development of the LIBSVM library for the support vector machine (SVM) technique [16] in Machine Learning. Though source codes of many existing general graph generators are publicly available, from our research, there appears to be no handy software toolkit for users to apply existing general graph generators easily to their applications. Moreover, there is no end-to-end platform such that users can easily integrate their newly developed general graph generators for a comprehensive performance evaluation with existing approaches.

1.2 Contribution

In this paper, we aim to address the four problems above and our principal contributions are summarized as follows.

- We give a comprehensive overview of existing general graph generators, including recently emerged deep learning-based approaches. We group existing methods into four categories based on their foundation techniques. For each category, we describe the key features of generators, representative approaches and summarize their main characteristics.
- We conduct systematic and comprehensive experiments to compare the performance of general graph generators. Specifically, 20 representative general graph generators in all categories are evaluated; 12 popular graph datasets and 17 representative evaluation metrics are deployed in experiments. We also provide easy-to-follow standard recommendations about how to select the general graph generator under different settings. We believe such a comprehensive experimental evaluation is beneficial to both scientific communities and practitioners.
- The experience and insights we gained throughout the study enable us to engineer a new algorithm, Scalable Graph Autoencoder (SGAE), which can achieve a satisfactory trade-off between the graph simulation quality and efficiency (scalability).
- We implement an end-to-end platform for researchers and practitioners such that not only can they apply a variety of existing general graph generators directly to their applications but can also easily integrate their own-built general graph generators for comprehensive performance comparison and analytics. We believe this will greatly benefit future research and the application of graph generators.

Roadmap. The rest of the paper is organized as follows. In Sect. 2, we formally define the problem and provide an overview of general graph generators evaluated in this paper. We then present the details of graph generators and introduce their optimizing targets in Sect. 3. In Sect. 4, we present some improvements for general graph generators. Comprehensive experimental results for general graph generators and an introduction to the toolkit we developed are presented in Sect. 5. Section 6 concludes the paper.

2 Background

2.1 Problem definition and notations

We define a graph $G = (V, E)$. V denotes a set of n nodes (vertices), and a set of m edges $E \subseteq V \times V$, where a tuple $e = (u, v) \in E$ represents an edge between two vertices u and v in V. The graph G can also be represented by an adjacency matrix $\mathbf{A} \in \{1, 0\}^{n \times n}$. As reflected in the literature, we assume G is an undirected graph, and hence the adjacency matrix of the graph is symmetric. Additionally, we denote the (optional) node-feature matrix associated with the graph as $\mathbf{X} \in \mathbb{R}^{n \times d}$ where n denotes the number of nodes and d denotes the dimension of the node feature. We denote the initiator matrix of the graph G by $\mathbf{M}_I$.

Problem Statement. Given a set of observed graphs $\{G\}$, a general graph generator aims to learn a generative model to capture the structural distribution of the graphs, such that a set of new graphs $\{G'\}$ with similar structural distribution can be generated.

Ideally, a general graph generator should be able to generate new graph which has exactly the same distribution as the observed graphs. Nevertheless, as discussed in Sect. 1, it is notoriously difficult to tell if two graphs are from the same distribution due to the complex nature of graph structure. In practice, we have to resort to representative evaluating metrics in the literature in our experiments, each of which aims to quantitatively capture the likelihood of two graphs (graph distributions) from one perspective (e.g., degree distribution). Please refer to Sect. 5 for more details. Hopefully, a good graph generator should be able to generate new graphs with the same distributions regarding all the above metrics. Moreover, the generating algorithm should be efficient and scalable such that the users can efficiently handle large-scale graph in real-life applications.

2.2 Scope

As shown in the recent survey [11], existing graph generators can be classified into two categories: general graph generators (e.g., [2,10,34,77]) and domain-specific graph generators (e.g., [6,7,30,60,80]). Notably, general graph generators aim to mimic the structures of the observed graphs so that the generated graph can reproduce such properties of the preserved graphs as degree distribution and the path length distribution regardless of the domains. Domain-specific graph generators consider particular domains such as semantic web (e.g., [30]), graph database (e.g., [6]), temporal graphs (e.g., [80]), and social networks (e.g., [7]).

To make a comprehensive yet focused comparison of graph generators, here we mainly consider *general graph generators*. In Sect. 2.3, we classify existing general graph generators into four categories based on their foundation technique. For each category, the key features and main characteristics, as well as a set of representative approaches, are introduced in detail in Sect. 3. To make a comprehensive performance evaluation, we include a large sample of representative graph datasets and evaluation metrics as observed in the literature covering experiments with general graph generators.

2.3 Classification of general graph generators

In this subsection, we classify general graph generators according to the key foundation techniques to allow readers to form a natural first acquaintance of these approaches. As demonstrated in Table 1, we distinguish (1) simple graph model-based general graph generator (*simple model-based generator* for short); (2) complex graph model-based general graph generator (*complex model-based generator* for short) (3) autoencoder-based general graph generator (*autoencoder-based generator* for short); and (4) generative adversarial network-based general graph generator (*GAN-based generator* for short). Below is a brief overview of each category of general graph generator. Table 1 also shows the inputs and optimization objectives of each generator.

1. **Simple model-based generator** These generators rely on well-known simple graph models such as the Binomial graph model [20], randomized small-world graph model [66], and preferential graph models [3], each of which can be expressed explicitly by formulas, given just a few parameters. We can generate a new graph directly by tuning the parameters of the simple graph model to preserve certain properties of observed graphs, such as the power-law degree distribution for the preferential graph model (e.g., [3]). In Table 1, we show 8 representative simple model-based generators. These require fewer parameters than other categories, resulting in relatively poor expressive power for observed graphs. To enhance the expressive power of the generator, we can consider the hierarchical combination of the simple models where a set of small "local" simple graphs are combined hierarchically to generate the "global" graph. For instance, the Kronecker graph model [37] gives a concrete example that a small graph's adjacency matrix $\mathbf{A}_{local} \in \{0, 1\}^{2 \times 2}$ can be extended recursively to $\mathbf{A}_{global} \in \{0, 1\}^{2^k \times 2^k}$.
2. **Complex Model-based Generator** By using sophisticated models with many more parameters, we can better capture the properties of the observed graphs. The complex model-based generators rely on probabilistic graph models or decision-process-based graph models such as mixed membership stochastic blockmodels [1], stochastic Kronecker graph model [37] and decision-

Table 1 Categories, names, reference, and optimization objectives of general graph generators

Category	Graph generator	Optimization objective
Simple Model-Based Generator	E-R [20]	None.
	W-S [66]	None
	B-A [3]	None
	RTG [2]	None
	BTER [35]	None
	SBM [27]	None
	DCSBM [31]	None
	R-MAT [15]	None
Complex Model-Based Generator	Kronecker [37]	Maximize the likelihood of permutation
	MMSB [1]	Maximize a posteriori probability of blocks from observed graphs
	GraphRNN [77]	Maximize the likelihood of permutations and edge dependence
	GRAN [42]	Maximize the likelihood of a family of permutations
	BiGG [18]	Maximize the likelihood of orderings of the binary tree
Autoencoder-Based Generator	VGAE [34]	Learn the generative distribution
	Graphite [23]	Learn the generative distribution
	SBMGNN [46]	Learn the sparse generative distribution
GAN-Based Generator	ARVGA [49]	Learn the generative distribution adversarially
	NetGAN [10]	Optimize the Wasserstein generative adversarial loss
	CondGEN [73]	Optimize the reconstruction loss adversarially

process-based graph models [18,42,77], each of which is modeled by a considerable number of parameters. These generators simulate a new graph by optimizing the likelihood function to fit the generative distribution of the observed graphs. In Table 1, we show 5 representative complex model-based generators.

3. **Autoencoder-based Generator** The autoencoder is a type of generative model used to learn a representation (encoding) and reconstruct input distribution (decoding) for a set of data in an unsupervised manner, which has been widely used for image and text data generation. In the field of autoencoder-based graph generators, i.e., graph autoencoders are usually parameterized by graph neural networks (GNN). For examples, VGAE [34], Graphite [23], and SBMGNN [46] are parameterized by graph convolutional neural networks. In Table 1, we show 3 representative autoencoder-based generators.
4. **GAN-based generator** The graph-generative adversarial models are based on generative adversarial network (GAN) architecture, which is used to learn a robust generative distribution for the observed data in the manner of a game between the *generator* and the *discriminator*. In the field of GAN-based graph generators, the discriminators remain flexible to allow control of the generative distribution of the generators' output, e.g., node latent variables [49], graph representations [73], and random walks [10]. The generator component can have a more robust output than autoencoder-based graph generators. In Table 1, we show 3 representative GAN-based generators.

2.4 A complimentary classification

For a better understanding the techniques of the representative graph generators, in this subsection, we provide a more complex and deep taxonomy as illustrated in Fig. 1, which consists of 5 categories and 17 sub-categories.

2.4.1 Sequential generating

means evolutionarily modeling a graph G; that is, relying on an existing incomplete graph to generate new elements. A graph G is represented by a sequence of elements, i.e., $S_N = \{s_1, ...s_N\}$. Then the new element is generated with $s_{N+1} = f(S_N)$, where f denotes the generative model (e.g., RNN, MLP). Following are three sub-categories:

1. **Node sequence.** GraphRNN [77] and [41,59,62] generate a graph through generate nodes and associated edges one-by-one. We select GraphRNN as representative of this category because it can generate graphs with more than 1000 nodes (compared with [41,59]) and has better performance and influence in generating general graphs (compared with [62]).
2. **Edge sequence.** BiGG [18] and [4,5] model a graph as a sequence of edges. We select BiGG as representative of

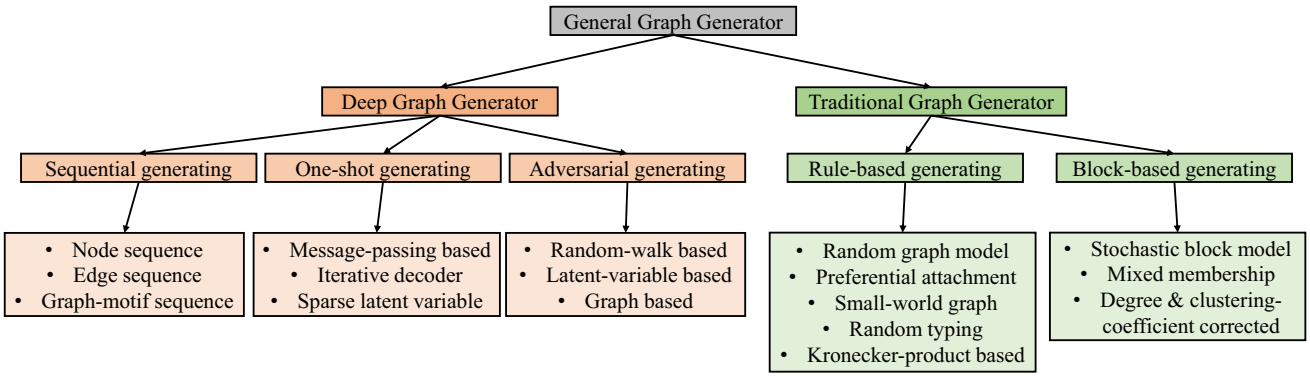

Fig. 1 Another classification of graph generators

this category because it achieves better performance and scalability in generating general graphs (compared with [4,5]).
2. **Motif sequence.** GRAN [42] and [29,51] generate graph motifs (e.g., a block of nodes and associated edges) sequentially to generate a complete graph. We select GRAN as representative of this category because it is more scalable than [29,51].

2.4.2 One-shot generating

means modeling an entire graph directly, that is, the elements on the graph are generated with no sequential dependencies. Corresponding generators can be further classified into 3 sub-categories:

1. **Message passing-based encoder.** VGAE [34] and [57] are proposed to generate one graph in one shot through message passing-based encoder. We choose VGAE as representative because it first proposed graph convolutional technology to generate graphs.
2. **Iterative decoder.** Graphite [23] generates graphs through reserve message-passing decoder. Graphite is selected as representative because of its influence and contribution on new technology, i.e., iterative decoder.
3. **Sparse latent variables.** SBMGNN [46] generates graphs through modeling sparse latent variables. SBMGNN is selected as representative because of its contribution on preserving community structure through sparse latent variables.

2.4.3 Adversarial generating

means training graph generator through a game between generator and discriminator. Corresponding models can be divided into the following 3 sub-categories:

1. **Random-Walk based.** NetGAN [10] and [81] generate random walks through adversarial training. We choose NetGAN as representative because it has better performance on generating general graphs.
2. **Latent-variable based.** ARVGA [49] is selected because of its contribution on adversarial training for latent variables.
3. **Graph based.** CondGEN [73] is selected because of its contribution on adversarial training for generated graphs.

2.4.4 Rule-based generating

means modeling a graph through explicit operations and a small number of parameters. This category's graph generators model graphs through selecting samples from predefined graph families [3,37,53]. They can be further classified into the following 5 sub-categories:

1. **Random graph model.** E-R [20] generate each edge by sampling from a Bernoulli distribution parameterized with a fixed value. We choose E-R, B-A, and W-S as representatives because of their influence.
2. **Preferential attachment graph model.** B-A [3] randomly add edges for new node to generate scale-free graphs.
3. **Small world graph model.** W-S [66] randomly rewrite edges from cycle graph to generate a small-world graph.
4. **Random typing graph model.** RTG [2] generates edge-list through random-typing process. We select RTG because of its influence and technical contribution.
5. **Kronecker-product based.** R-MAT [15], Kronecker [37] and [47] model a graph through a regressive dropping edges mechanism similar to Kronecker-product. We select R-MAT and Kronecker as representatives because of their influence.

2.4.5 Block-based generating

means modeling a graph through modeling "blocks," i.e., a subset of nodes. Corresponding models can be classified into the following 3 sub-categories:

1. **Vanilla stochastic blockmodel** SBM [27] was first proposed to model social networks.
2. **Mixed membership.** MMSB [1] models a graph with mixed membership of blockmodel. MMSB provides a generalization of SBM to have better quality of simulating graphs.
3. **Degree & clustering-coefficient corrected.** DCSBM [31] was proposed to correct the blockmodel with observed degree distributions. BTER [35] was proposed to correct the average clustering coefficient in each block, and correct the degree distribution through a two-level edge sampling process.

3 General graph generators

In this section, we describe the key features of general graph generators in each category and summarize their main characteristics. For each category, we present the details of its representative generators.

3.1 Simple model-based generator

Simple model-based generators rely on some well-known simple graph models (families), such as Binomial graphs [20], randomized small-world graphs [66] and preferential attachment graph model [3]. Usually, each family of graph models can capture one or a few properties of the real-life graphs well. For instance, the B-A model can generate free-scale networks, and the W-S model can capture the key characteristics of the small-world graphs. Below are details of 8 representative techniques in this category.

E-R. Binomial graph model (E-R) was first proposed and studied by Erdős et al [20]. Each edge of the binomial graph is generated independently with the probability $P(\mathbf{A}_{i,j} = 1) = p$, where the constant p is controlled by the user. Given an undirected and no self-loop graph with n vertices and m edges, the parameter p is immediately available with $p = \frac{2m}{n(n-1)}$. The naive implementation of E-R-based graph generator generates an edge with probability p for each pair of vertices, with time complexity $O(n^2)$ and space complexity $O(m+n)$. As shown in [8], an efficient algorithm with time complexity $O(m+n)$ is used in practice.

W-S. Small-world (W-S) graph model was first proposed by Watts and Strogatz [66]. It can simulate the random connection of real graphs by re-sampling edges and make the distribution of the generated graph between the E-R graph and completely regular graph by adjusting the sampling probability and the number of connected edges. The implementation of W-S-based graph generator randomly resets the target nodes of each source node from a regular circle graph, having a time complexity $O(k \times n)$ and space complexity $O(m+n)$, where k is the number of edges for each source node.

B-A. The preferential-attachment [3] (B-A) graph model needs to generate graph nodes sequentially and add a fixed number of edges to new nodes to generate a scale-free graph with the power-law distribution. The implementation of a B-A-based graph generator has the time complexity of $O(k \times n)$ and the space complexity of $O(m+n)$, where k is the number of edges a new node will attach.

RTG. The Random-Typing Generator [2] (RTG) creates a two-dimension (2d) keyboard to type words that are represented as edges of a graph. The graph generation process is modeled as the process of typing words. There are four parameters (K, W, q, β) to control the generative distribution. K is the number of possible characters in one word, and W is the number of words. q is the probability of typing space. β controls the probability of randomly typing diagonal characters and non-diagonal characters on the 2d keyboard. These parameters help generate graphs with power-law degree distribution and communities. Moreover, RTG can also generate bipartite graphs. The implementation of the RTG-based graph generator requires $O(m \log n)$ time complexity and $O(m+n+K^2)$ space complexity.

BTER. The Block Two-level Erdős-Rényi model [35] (BTER) was proposed to simulate both the realistic graph's degree distribution and its clustering coefficient by degree. These two elements are extracted directedly from observed graphs. BTER creates affinity blocks first, to assign degrees to nodes. Then BTER assigns edges in and between affinity blocks to match the degree distribution. The clustering coefficient per degree is assigned when creating links between nodes in the same affinity blocks. The implementation of the BTER-based graph generator has a time complexity $O(m+n)$ and space complexity $O(m+n)$.

SBM & DCSBM. The Stochastic Block Model [27] (SBM) generates a random graph based on the probability matrix of the block $\mathbf{B}$ and the number of nodes of each block. The SBM-generated graph is considered as a connected set of E-R graphs with community structure. The Degree-corrected Stochastic Block Model [31] (DCSBM) sets the degree of each node following the Stochastic Block Model, the process thereby tuning the degree distribution of the generated graph. To obtain the block probability matrix $\mathbf{B}$, we choose the best block partition by maximizing the modularity [9] of the observed graphs, with linear time complexity of m.

After the number of blocks and the probability of generating edges between blocks are determined, the time complexity for generating one graph of SBM and DCSBM is $O(n^2)$ in their naive implementations. As shown in [31], an efficient implementation of the (DC) SBM-based graph generator has a $O(m+n)$ time complexity and $O(m+n)$ space complexity.

R-MAT. The recursive graph model [15] (R-MAT) divides the adjacency matrix recursively and determines a graph by dropping edges into one quadrant of four recursively. As for the parameter optimization of the R-MAT model, we use the empirical parameters mentioned by the author as the initial matrix $\mathbf{M}_I$. R-MAT generates one graph with n nodes and m edges by randomly sampling edges with time complexity $O(m\log(n))$ and space complexity $O(m+n)$. R-MAT also inspired Dai et al. [18] to compress one row of adjacency matrix into a binary tree.

3.2 Complex model-based generator

With more parameters and complex model architecture, we know statistics learning and parameterized models can simulate and generate the adjacency matrix of one graph numerically [1,37]. Although there are $n!$ permutations of adjacency matrices to represent the same graph, nevertheless they successfully achieve better simulation quality than simple model-based generators. In addition to being represented as an adjacency matrix, the graph structure can also be modeled as a decision-making process. In this process, nodes and edges are generated sequentially, which is easy to represent and learn for regular graphs. Below we introduce 5 representative complex model-based general graph generators.

MMSB. The Mixed Membership Stochastic Blockmodel [1] uses a probabilistic model to build a graph. First, the mixed membership vector π_i of the node i is sampled from the Dirichlet distribution with $\pi_i \sim \text{Dirichlet}(\alpha)$, where α denotes the parameters. Then the indicator of the edge is obtained by sampling from a multinomial distribution with $z_{i,j} \sim \text{Multinomial}(\pi_i)$. Finally, the edge is sampled from the Bernoulli distribution, the probability of which is obtained by a bilinear function of edge indicators with $\mathbf{A}_{i,j} \sim \text{Bernoulli}(z_{i,j}^T \mathbf{B} z_{i,j})$, where $\mathbf{B} \in \mathbb{R}^{K\times K}$ represents the probability of generating one edge between two blocks.

The MMSB maximizes a-posteriori to approximate its parameters, but its hyperparameter K, the number of blocks, is difficult to select when simulating large graphs. We use a *Louvain* community detection algorithm [9] to select the number of blocks K. Because the MMSB aims to approximate the observed adjacency matrices, it has a time complexity of $O(n^2)$ and space complexity of $O(n^2)$ to update its parameters. Note that the inference algorithm of MMSB can easily be paralleled.

Kronecker. The Kronecker graph model uses the Kronecker product to build a graph, where the initiator graph is self-connected and has a binary matrix to represent the edges of the graph. The initiator matrix of the stochastic Kronecker graph is not binary, which denotes the probability of generating one edge. Different from R-MAT, the sum of the initiator is not 1 and the probability of each edge can be calculated independently as follows:

$$P(\mathbf{A}_{i,j}=1) = \prod_{k=0}^{|\log n|-1} \mathbf{M}_I[\lfloor \frac{i-1}{2^k} \rfloor \pmod 2 + 1, \lfloor \frac{j-1}{2^k} \rfloor \pmod 2 + 1]. \tag{1}$$

In the optimization stage, the Kronecker graph model looks for the best node permutations and uses the maximum likelihood principle to estimate the parameters of the initiator matrix. To generate graphs faster, the Kronecker model imitates R-MAT to throw edges recursively into the adjacency matrix, which accelerates the generation process of the stochastic Kronecker graph, with the time complexity $O(mlog(n))$ and space complexity $O(m+n)$.

GraphRNN. Two recurrent neural networks (RNN) are deployed by GraphRNN. The first, called *graph-level* RNN, is used to store generated nodes and to generate new nodes. The second, called *edge-level* RNN, is used to store generated edges on the newly generated node and infer new edges. Each edge is sampled from the Bernoulli distribution, which is parameterized by the output of the *edge-level* RNN. At this point, the graph generation is modeled as a decision sequence. Assuming that h_0 and $h_{i,0}$ represent the initial graph state and node i's hidden state, respectively, GraphRNN's generation process can be formulated as follows:

$$\begin{aligned} h_i &= \text{RNN}_1(h_{i-1}, \mathbf{A}_{i-1}), \\ \theta_{i,j} &= \text{RNN}_2(h_{i,j-1}, \mathbf{A}_{i,j-1}), \quad (h_{i,0} = h_i) \\ \mathbf{A}_{i,j} &\sim \text{Bernoulli}(\theta_{i,j}), \quad (j < i) \end{aligned} \tag{2}$$

where $\mathbf{A}_{i-1}$ and $\mathbf{A}_i$ denote the adjacency vectors of the last node and next generated node, respectively. In the GraphRNN, Gated Recurrent Unit [17] (GRU) is used to encode the graph state and infer node-adjacency vectors.

For graph generation with no edge dependence, authors propose a variant named GraphRNN-S which replaces the second RNN with a multi-layer perceptron (MLP). Then the adjacency vector $\mathbf{A}_i$ will be sampled from a multivariate Bernoulli distribution parameterized by θ_i. The generation process of GraphRNN-S can be formulated as follows:

$$\begin{aligned} h_i &= \text{RNN}(h_{i-1}, \mathbf{A}_{i-1}), \\ \theta_{i,j} &= \text{MLP}(h_i), \\ \mathbf{A}_i &\sim \text{Bernoulli}(\theta_i) \end{aligned} \tag{3}$$

The variant performs well in generating protein graphs and other real-world graphs, which show less edge dependence than grid and community graphs. GraphRNN-S also promoted subsequent works [18,42] to scale the complex model-based graph generator up to larger graphs.

After modeling the graph as a sequence of decisions, the most important problem is how to prevent the order of nodes from affecting the generalization performance of the model. GraphRNN utilizes Breadth-first Search (BFS) ordering to reduce the number of permutations of the observed graphs and then maximizes the likelihood of the edge dependence and node permutations. It generates the nodes and edges of the graph step-by-step and requires $O(n^2)$ time complexity in each epoch. Thus, the complexity for training and inference is $O(e \times n^2)$ and $O(n^2)$, respectively, where e is the number of epochs used in training. Due to the necessary dependence on nodes and edges, the generation process of complete GraphRNN cannot be carried out in parallel. Therefore, the follow-up work is devoted mainly to improving the scalability of GraphRNN.

GRAN. Graph Recurrent Attention Networks [42] (GRAN) aims to improve the performance of GraphRNN by addressing the following issues: (1) low generalization due to strong dependence on node orderings and edges; (2) expressive capability of only one canonical node ordering; and (3) poor parallelization compared with a graph autoencoder. To reduce its dependence while retaining the expressiveness of the graph auto-regressive model (e.g., GraphRNN), GRAN leverages graph attention networks [65] (GAT) to infer the parameters of Bernoulli distributions when generating the whole graph. GRAN uses t steps to generate a graph, and each step generates B nodes, called a block. If $B > 1$, the generation process can accelerate by commencing the next block in the S-th row of the last generated block, called stride ($1 \leq S \leq B$). At t-th generation step, GRAN reduces the embedding size of previous nodes to generate large graphs by a linear mapping:

$$\begin{aligned} L_i &= [L_{b_i,1} ... L_{b_i,B}], \\ h_i &= \mathbf{W} L_i + b, \quad \forall i < t \end{aligned} \tag{4}$$

where $[\cdot]$ denotes a concatenation operation of vectors and $L_i \in \mathbb{R}^{Bn}$ is a vector concatenated by the output vectors of a block of nodes. The initial node representations are updated regressively with $h_i = \text{GRU}(h_i, \text{GAT}(h_i))$, as with [40]. After updating node representations, to express the edge dependences in one block, GRAN models the probabilities of generating edges through a mixture of Bernoulli distributions:

$$\begin{aligned} p(L_{b_t} | L_{b_1}, ..., L_{b_{t-1}}) &= \sum_{k=1}^{K} \alpha_k \prod_{i \in b_t} \prod_{j \leq i} \theta_{k,i,j}, \\ \alpha_1, ..., \alpha_K &= \text{Softmax}(\sum_{i \in b_t, j \leq i} \text{MLP}_\alpha(h_i - h_j)), \\ \theta_{1,i,j}, ..., \theta_{K,i,j} &= \text{Sigmoid}(\text{MLP}_\theta(h_i - h_j)) \end{aligned} \tag{5}$$

where K is the number of mixture components. When $K > 1$, the edges generated in parallel are no longer independent because of the latent mixture components, which maintains the edge dependence without loss of parallelization.

To learn the graph-generative model under more than one canonical node orderings, GRAN proposes a new objective to maximize a lower bound of log-likelihood as follows:

$$\log p(G) = \log \sum_{\pi} p(G, \pi) \geq \log \sum_{\pi \in \Omega} p(G, \pi) \tag{6}$$

where Ω denotes the selected canonical orderings of the graph node. The greater the quantity of canonical orderings picked, the tighter the bound will be.

Inspired by the parallelism of graph neural networks (GNN), GRAN provides a flexible trade-off between computational cost and generative performance through adjusting the block size and stride length S, so that it requires a time complexity of $O(\frac{n^2}{S})$ in each epoch. Its efficiency and scalability are significantly better than GraphRNN when generating large graphs, e.g., graphs with more than 500 nodes under our experiment setting.

BiGG. Inspired by the recursive graph model [15] (R-MAT), BiGG is a graph auto-regressive model with a tree structure. Assuming that G is a large sparse graph ($m \ll n^2$), generating only the edges of G is a more efficient choice than generating each node's adjacency vector and can be formulated as follows:

$$p(\mathbf{A}) = p(e_1) p(e_2 | e_1) ... p(e_m | e1, ..., e_{m-1}) \tag{7}$$

where each edge $e_i = (u, v)$ includes the indices of two nodes. Therefore, the generation process contains m steps. In previous work, a single edge can be factorized with $p(e_i) = p(u)p(v|u)$ and $p(v|u)$ is assumed to be simple multinomials over n nodes, which will result in the complexity of $O(n)$. BiGG reduces the number of decisions of specifying v through formulating $p(v|u)$ as follows:

$$p(v|u) = \prod_{i=1}^{\lceil \log_2 n \rceil} p(x_i = x_i^v) \tag{8}$$

where $x_i^v \in \{\text{left}, \text{right}\}$ denotes the i-th decision in the sequence of node v and $p(x_i = x_i^v)$ denotes the probability of the i-th decision leading to v. Note that $x_i^v =$ left (right) means the left (right) sub-tree is chosen in the i-th decision of node v. BiGG uses $E_u = \{(u, v) \in E\}$ to represent the set of edges connecting node u and $\mathcal{N}_u = \{v|(u, v) \in E_u\}$ to represent the set of neighbor nodes of u. For each of node u's row of adjacency matrix, generating all node u's edges E_u is equivalent to generating a node u's binary tree $\mathcal{T}_u$, where for each $v \in \mathcal{N}_u$ the generation process starts from the root node and ends in a leaf node. Each node t is generated with its left subtree $\text{lch}(t)$ and previously generated nodes as conditioning. The right subtree $\text{rch}(t)$ is generated after generating the left subtree and its dependencies, similar to the in-order traversal of the binary tree. Let $\text{context}_u(t)$ and $\text{context}_u(\text{lch}(t))$ represent the previous context and the summary context of the left subtree of node t, respectively. Then the recursively generated $p(E_u)$ can be formulated as follows:

$$\begin{aligned} p(E_u) =& p(\mathcal{T}_u) \\ =& \prod_{t \in \mathcal{T}_u} p(\text{lch}(t)) p(\text{rch}(t)) \\ =& \prod_{t \in \mathcal{T}_u} p(\text{lch}(t)|\text{context}_u(t)) \\ & p(\text{rch}(t)|\text{context}_u(t), \text{context}_u(\text{lch}(t))). \end{aligned} \tag{9}$$

where $p(\text{lch}(t)|\cdot)$ and $p(\text{rch}(t)|\cdot)$ are Bernoulli distributions parameterized by TreeLSTM networks [63]. So far, each row of adjacency matrix $\mathbf{A}$ can be generated recursively through the construction of binary tree and $p(E) = \prod_{u=1}^{n} p(E_u)$ costs $O(m \log n)$ time. The full model is going to generate the adjacency matrix row by row. Similarly, BiGG models the root nodes of n *edge-binary* trees as the summary context of nodes. It also models the summary context into a *row-binary* tree recursively, which costs $O(n \log n)$ time. BiGG generates a graph requiring $O((m + n) \log n)$ time complexity, which is especially efficient when generating large sparse graphs. Each depth of parameters in the binary tree can be updated in parallel, resulting in $O(\log n)$ steps, which is more efficient than $O(n)$ steps in GRAN and GraphRNN-S.

3.3 Autoencoder-based generator

The complex model-based graph generators encountered a bottleneck of generative performance until the advent of the graph autoencoders (GAEs). Thanks to the progress of variational autoencoder and deep learning [32,52], researchers extend autoencoder to the field of graph representation learning and graph generation. Meanwhile, graph neural networks [33,65,69,78] are also proven to be successful on graph representative and generative models.

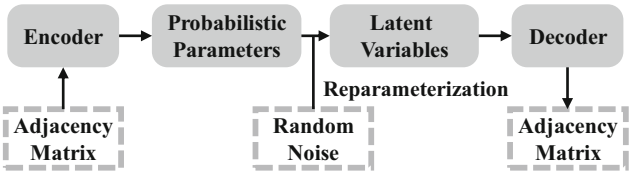

Fig. 2 A summary of autoencoder-based generators

Figure 2 illustrates the framework of the autoencoder-based generators. Specifically, an encoder is used to learn the representation (i.e., encoding) of the observed graphs, and the probabilistic parameters are captured by a deep graph neural network (GNN). By adding some random noise, we can reparameterize the latent variables and reconstruct (i.e., decode) a new graph with the decoder. The parameters of GNNs will be updated according to the accuracy of the simulation. The above process is repeated till a new graph with high quality (e.g., small reconstruction error) can be generated. Below, we introduce three representative autoencoder-based graph generators.

VGAE. The variational graph autoencoder [34] was proposed by Kipf et al. to naturally extract node features and infer the generative distribution of observed graphs. VGAE uses a GNN [33] as the encoder of graph data. The GNN layer parameterized by $\mathbf{W}$ is defined as follows:

$$\text{GNN}_{\mathbf{W}}(\mathbf{X}, \mathbf{A}) = \bar{\mathbf{D}}^{-\frac{1}{2}} \bar{\mathbf{A}} \bar{\mathbf{D}}^{-\frac{1}{2}} \mathbf{X} \mathbf{W} \tag{10}$$

where $\bar{\mathbf{A}} = \mathbf{A} + \mathbf{I_n}$ denotes the adjacency matrix with an added self-loop and $\bar{\mathbf{D}}$ is degree matrix of $\bar{\mathbf{A}}$ with $\bar{\mathbf{D}}_{i,i} = \sum_j \bar{\mathbf{A}}_{i,j}$. The VGAE uses two layers of GNN to infer the parameters of the stochastic variables $\mathbf{Z} \in \mathbb{R}^{n \times f}$, where f denotes the dimension of latent variables. For the GAE model, the inference model of $q(\mathbf{Z}|\mathbf{X}, \mathbf{A})$ is parameterized as follows:

$$\mathbf{Z} = \text{GNN}_{\mathbf{W_2}}(\sigma(\text{GNN}_{\mathbf{W_1}}(\mathbf{X}, \mathbf{A}))) \tag{11}$$

where σ denotes the nonlinear activation function. Then the generative model is defined as a bilinear function of nodes' latent variables with $P(\mathbf{A}_{i,j} = 1) = \sigma(\mathbf{Z}_i \mathbf{Z}_j^T)$, where $\mathbf{Z}_i$ denotes the latent variables of node i. VGAE has two optimization objectives, one is to reconstruct the adjacency matrix and the other is to approximate the a priori distribution. That leads to optimizing the variational lower bound as follows:

$$L = E_{q(\mathbf{Z}|\mathbf{X},\mathbf{A})}[\log p(\mathbf{A}|\mathbf{Z})] - KL[q(\mathbf{Z}|\mathbf{X}, \mathbf{A})||p(\mathbf{Z})] \tag{12}$$

where $p(\mathbf{A}|\mathbf{Z})$ is the generative distribution with $p(\mathbf{A}|\mathbf{Z}) = \prod_i \prod_j p(\mathbf{A}_{i,j}|\mathbf{Z}_i, \mathbf{Z}_j)$, $KL(\cdot||\cdot)$ denotes the Kullback-Leibler divergence measuring the distance between two distributions,

and $p(\mathbf{Z})$ is a Gaussian prior with $p(\mathbf{Z}) = \prod_{i=1}^{n} \mathcal{N}(\mathbf{Z}_i|0, \mathbf{I})$. VGAE needs $O(n^2)$ time to generate a new graph with space $O(m+n)$, and it takes $O(n^2)$ time in each epoch of the training process.

Graphite. Working similarly to VGAE, the iterative generative model of graphs [23] (Graphite) parameterizes the variational autoencoders with graph neural networks. The main contribution of Graphite is that it replaces the inner-product decoder of VGAE with node representations and intermediate graphs. The decoding process is formulated as follows:

$$\mathbf{Z}^* = \text{GNN}_\theta(\hat{\mathbf{A}}, [\mathbf{Z}|\mathbf{X}]), \text{ with } \hat{\mathbf{A}} = \frac{\mathbf{Z}\mathbf{Z}^T}{||\mathbf{Z}||^2} + \{1\}^{n\times n} \quad (13)$$

where θ denotes the parameters of GNN and $[\cdot|\cdot]$ means a concatenation operation. $\hat{\mathbf{A}}$ is an intermediate graph, to which is added a constant of 1 to keep the matrix non-negative. The feature matrix $\mathbf{Z}^*$ can be refined gradually until getting the final features to generate an adjacency matrix.

Graphite is consistent with VGAE in encoder and optimization objectives. It still has a time complexity $O(n^2)$ in each epoch and space complexity $O(m+n)$ due to the inner product involved although the implementation of the iterative decoder is accelerated to $O(n \times f^2)$, where f is the dimension of the latent features.

SBMGNN. Nikhil et al. [46] proposed a *sparse* variational autoencoder for graphs by merging the interpretability of SBM and the fast inference of graph neural networks. As in MMSB, SBMGNN uses a stick-breaking construction of the Indian Buffet Process [64] to infer the size of community memberships, which is formulated as follows:

$$\begin{aligned} v_k &\sim \text{Beta}(\alpha, 1), \quad k = 1, ..., K \\ b_{nk} &\sim \text{Bernoulli}(\pi_k), \pi_k = \prod_{j=1}^{k} v_j \end{aligned} \quad (14)$$

where K is the number of communities and π_k is the probability of all memberships. Unlike MMSB, SBMGNN infers the variational parameters of these distributions through graph neural networks. SBMGNN also uses the variational graph autoencoder (VGAE) to approximate the dense latent variables r_n. In contrast to VGAE, SBMGNN models the node embeddings as $z_n = b_n \odot r_n$ with remaining other sections consistent. So far, the inference process can be defined as follows:

$$\begin{aligned} q_\phi(v_{nk}) &= \text{Beta}(v_{nk}|\text{GNN}_\alpha(\mathbf{X}, \mathbf{A}), \text{GNN}_\beta(\mathbf{X}, \mathbf{A})) \\ q_\phi(b_{nk}) &= \text{Bernoulli}(b_{nk}|\text{GNN}_\pi(\mathbf{X}, \mathbf{A})) \\ q_\phi(r_n) &= \mathcal{N}(\text{GNN}_{\mu_n}(\mathbf{X}, \mathbf{A}), \text{diag}(\text{GNN}_{\sigma_n^2}(\mathbf{X}, \mathbf{A}))) \end{aligned} \quad (15)$$

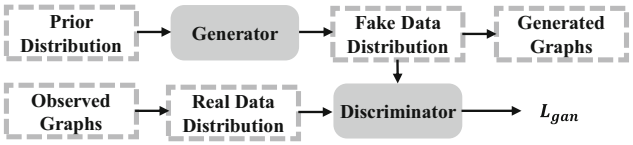

Fig. 3 A summary of generative adversarial network (GAN)-based generators

The graph generation process of SBMGNN is the same as other autoencoder-based graph generators. The overall optimization objective of this inference and generative model can be formulated as the sum of KL divergence of these approximating distributions and the reconstruction loss, which can be extended from the objective of VGAE. SBMGNN has a time complexity of $O(n^2)$, and has realized the model's interpretability with the cost of more model parameters.

3.4 GAN-based Generator

The core of generative adversarial networks [21] (GANs) is to use a discriminator to generate fake data with good quality and robustness. As shown in Fig. 3, the key idea of the GAN-based graph generators is to use the graph generator to establish the mapping from the random variable to the fake hidden variable of the graph, and put it into the discriminator with the encoded hidden variable of the real graph. The objective of these models is to make the generator generate with stability and produce realistic hidden variables, which can be decoded to simulate the realistic graphs. Below are three representative GAN-based generators.

ARVGA. The adversarial regularized variational graph autoencoder [49] (ARVGA) was proposed to generate embeddings of the graph. Given a graph G, the hidden variable matrix $\mathbf{Z}$ is obtained by using the same method of graph encoding used in VGAE. Then the discriminator will repeatedly update its parameters by optimizing following cross-entropy cost:

$$\mathbb{E}_{p(z)\sim q(\cdot|\mathbf{Z})} \log(1 - D(z)) + \mathbb{E}_{p(a)\sim\mathcal{N}(\cdot|\mathbf{0},\mathbf{I})} \log D(a) \quad (16)$$

where z, a are the sample vectors from $\mathbf{Z}$ and real data distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$, respectively, and D is built on a standard multi-layer perceptron (MLP). Before each update of parameters of the graph autoencoder, the parameters of the discriminator are updated for multiple times.

The main difference between ARVGA and VGAE is that the former regularizes the output of the encoder directly into a priori distribution through a discriminator, not just through KL divergence to approximate a priori distribution. The generated robust embeddings are proved to have better performance on link prediction and node clustering than VGAE. The time complexity of ARVGA is $O(n^2)$ in each

epoch. Thus, the complexity of training and graph inference is $O(e \times n^2)$ and $O(n^2)$, respectively. Recall that e denotes the number of epochs required in the training process.

NetGAN. NetGAN [10] is the first model to generate graphs through random walks. It leverages long short-term memory [26] (LSTM) to generate random walk sequences. The longer the sequence length of random walks, the more topology information is captured by LSTM. Note that when the sequence length is 2, NetGAN will directly learn the edge probabilities.

As the input of NetGAN's generator component, the initial cell state C_0 and the initial hidden state h_0 of NetGAN are mapped from a multivariate normal distribution as follows:

$$\begin{aligned} &z \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \\ &C_0 = \text{MLP}_{(C)}(z), \quad h_0 = \text{MLP}_{(h)}(z) \end{aligned} \tag{17}$$

where $\text{MLP}_{(\cdot)}$ consists of two linear layers and *tanh* activation. Then LSTM parameterized by θ can start inferring the random walks as follows:

$$\begin{aligned} (p_1, C_1, h_1) &= \text{LSTM}(C_0, h_0, \mathbf{0}), \\ v_1 &\sim \text{Cat}(\text{Softmax}(p_1)), \\ (p_T, C_T, h_T) &= \text{LSTM}(C_{T-1}, h_{T-1}, v_{T-1}), \\ v_T &\sim \text{Cat}(\text{Softmax}(p_T)), \end{aligned} \tag{18}$$

where Cat denotes a categorical distribution. So far, the generator $\mathcal{G}$ can sequentially generate random walks $(v_1, \ldots, v_T)$. However, p_T and v_T have a dimension of n, resulting in a high computation cost in LSTM. Therefore, an up-project matrix $\mathbf{W}_{up} \in \mathbb{R}^{h \times n}$ is used to map the output of LSTM $o_T \in \mathbb{R}^h$ into $\mathbb{R}^n$. A down-project matrix $\mathbf{W}_{down} \in \mathbb{R}^{n \times h}$ is used to map node v_t into a low-dimensional input of LSTM. Generated random walks and real random walks are fed into the discriminator $\mathcal{D}$ parameterized by another LSTM, which outputs a probability of the random walk's being real. The model parameters are trained through Wasserstein GAN [24] (WGAN) framework.

After updating the parameters of the generator $\mathcal{G}$, inferred random walks through $\mathcal{G}$ can be decoded as new graphs through assembling the adjacency matrix. A score matrix $\mathbf{S}$ represents the appearance probability of each edge in generated random walks. Then each edge (i, j) is sampled from the categorical distribution parameterized by $p_{(\cdot,\cdot)}$ with $p_{(i,j)} = \frac{s_{i,j}}{\sum_{u,v} s_{u,v}}$. The edges of the whole graph are generated by selecting the top m entries of the score matrix. Because of the edge-sampling strategy of NetGAN, the graph generation process has a complexity of $O(n^2)$, and the numbers of nodes and edges are fixed to n and m, respectively.

CondGEN. The conditional variational autoencoder with a generative adversarial network (CondGEN) was proposed in [73]. The encoding and generation of the conditional graph structure are also considered. Permutation-invariance and generating arbitrary size of graphs are the main contributions of CondGEN and are achieved by modifying the derivation of stochastic latent variable $\mathbf{Z}$ of VGAE below:

$$\begin{aligned} \bar{\mu} &= \frac{1}{n} \sum_{i=1}^{n} g_\mu(\mathbf{X}, \mathbf{A})_i, \\ \bar{\sigma}^2 &= \frac{1}{n^2} \sum_{i=1}^{n} g_\sigma(\mathbf{X}, \mathbf{A})_i^2, \\ q(z_i|\mathbf{X}, \mathbf{A}) &\sim \mathcal{N}(\bar{z}|\bar{\mu}, \text{diag}(\bar{\sigma}^2)) \end{aligned} \tag{19}$$

where $g(\mathbf{X}, \mathbf{A}) = \text{GNN}_{\mathbf{W}_2}(\text{ReLU}(\text{GNN}_{\mathbf{W}_1}(\mathbf{X}, \mathbf{A})))$ is a two-layer GNN model. The modeling of $\bar{z}$ is essential for preserving permutation-invariance and can be regarded as the *graph embedding* of the graph G. Samples from $\bar{z}$ can also be decoded into a new graph through an FNN-based decoder. For the adversarial optimizing objectives, different from ARVGA, CondGEN is designed to learn the generative distribution of observed data as follows:

$$\begin{aligned} L_{gan} = &\log(\mathcal{D}(\mathbf{A})) + \log(1 - \mathcal{D}(\mathcal{G}(\mathbf{Z}_p))) + \\ &\log(1 - \mathcal{D}(\mathcal{G}(\mathbf{Z}_q))) \end{aligned} \tag{20}$$

where $\mathcal{D}$ is a two-layer GNN followed by a two-layer FNN and $\mathbf{Z}_p$ and $\mathbf{Z}_q$ are the latent variables sampled from both the Gaussian prior and the latent variable distribution $q(z_i|\mathbf{X}, \mathbf{A})$, respectively. CondGEN is designed to learn the structure distributions of a set of graphs, then generate permutation-invariant graphs, which can be controlled by the corresponding conditions. Due to the spectral embeddings' derivation in CondGEN, it has a training time complexity $O(n^3)$ at each epoch. In our experiment, we leverage CondGEN to generate new graphs without any encoding process. Thus, CondGEN has an inference time complexity $O(n^2)$.

3.5 Summary

In this subsection, we provide a summary of graph generators in terms of time complexity and space complexity. We also evaluate important properties of graph generators including *permutation invariance* and *community preserving*.

Time Complexity In Table 2, we report the time complexity of each generator for the learning (training) process and the new graph inference process. Generally, there is no training process for the simple model-based generators and they can easily calculate the desired parameters with time complexity $O(m + n)$. Due to the simplicity of the model, its inference time is very efficient as well. It takes much more learning and inference time for generators from other categories. A dominant cost of many graph generators is the generation of the

Table 2 Complexity, scalability, and permutation invariance of general graph generators.

Graph generator	Training time complexity per Epoch	Inference Time complexity	Space Complexity	Permutation	Invariance	Preserving	community
E-R [20]	–	$O(m+n)$	$O(m+n)$	✓			
W-S [66]	–	$O(nk)$	$O(m+n)$	✓			
B-A [3]	–	$O(nk)$	$O(m+n)$				
RTG [2]	–	$O(m \log n)$	$O(m+n+K^2)$				
BTER [35]	–	$O(m+n)$	$O(m+n)$				
SBM [27]	–	$O(m+n)$	$O(m+n+B^2)$			✓	
DCSBM [31]	–	$O(m+n)$	$O(m+n+B^2)$			✓	
R-MAT [15]	–	$O(m \log n)$	$O(m+n)$	✓			
MMSB [1]	$O(n^2)$	$O(n^2)$	$O(n^2)$			✓	
Kronecker [37]	$O(m \log n)$	$O(m \log n)$	$O(m+n+\log n)$	✓			
GraphRNN [77]	$O(n^2)$	$O(n^2)$	$O(n^2)$				
GRAN [42]	$O(n^2)$	$O(n^2)$	$O(m+n)$				
BiGG [18]	$O(n^2)$	$O(n^2)$	$O(m+n+\log n)$				
VGAE [34]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓		✓	
Graphite [23]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓		✓	
SBMGNN [46]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓		✓	
ARVGA [49]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓		✓	
NetGAN [10]	$O(n^2)$	$O(n^2)$	$O(n^2)$	✓		✓	
CondGEN [73]	$O(n^3)$	$O(n^2)$	$O(m+n)$	✓			

n and m are the amount of nodes and edges, respectively. k is the amount of edges attached to the previous node. K is the number of characters in the keyboard. B is the amount of blocks. m is the amount of edges. S is the length of stride

adjacency matrix, leading to a time complexity $O(n^2)$ in the inference process. For deep neural network-based generators, the learning time is also determined by the structure of the networks and the number of epochs. Particularly, GNN and RNN are two types of deep neural networks used by existing general graph generators, with time complexity $O(m+n)$ and $O(n^2)$, respectively, at each epoch. Due to the spectral embeddings' derivation in CondGEN, it has time complexity $O(n^3)$ at each epoch of the training process. We remark that the total training time also relies on the number of epochs required. In our experiments, GraphRNN takes much more training time compared to CondGEN due to the former's greater number of epochs involved in the training process. Moreover, the practical performance of the graph generators also depends on whether they can be easily paralleled in the system.

Space complexity Table 2 reports the space complexity of each generator. Generally, several generators are very space-efficient as they only need to keep the observed graphs with space $O(m+n)$. Note that for deep neural network-based generators, we assume the dimensionality of the latent variables (i.e., embeddings) of vertices is a constant (usually 32, 64 or 128 in practice). For graph neural networks (GNNs), we store the adjacency matrix and identity matrix as a sparse matrix, which costs $O(m+n)$ instead of $O(n^2)$. Thus, the corresponding space of the GNN-based generator is $O(m+n)$. For general recurrent neural networks (RNNs), such as GraphRNN-S and GRAN, we store the long-term memory of a sequence of nodes, with a space consumption dependent on the number of nodes. Thus, including the observed graphs, the corresponding space of the RNN-based generator is $O(m+n)$. It is also shown that MMSB, GraphRNN, and NetGAN are the most space-consuming generators because MMSB and GraphRNN need to maintain the probabilistic graph for all n nodes, while NetGAN needs to assemble a score matrix with $O(n^2)$ space complexity.

Permutation invariance property Table 2 shows the generators with the permutation invariance property. The permutation invariance property implies that we can set an arbitrary order of graph nodes for the learning process, and it has no effect on the simulation and generation results of the graph generator. Graph generators with the permutation invariance property can generalize to large graphs without considering the order of the nodes, which may come up with $O(n!)$ possible instances.

Community preserving property Table 2 also indicates the generators with the community preserving property. Graph generators need to preserve community structures of the observed graphs well. For instance, simulated graphs with similar community structures can be utilized to enhance community detection.

4 Improvement

The experience and insights gained from this study enable us to engineer a new method, namely Scalable Graph Autoencoder (SGAE), which can achieve a good trade-off between graph simulation quality and efficiency (scalability).

Our proposed method follows the framework of VGAE [34], with new techniques in the following three aspects.

Encoder We first leveraged Graph Neural Networks (GNNs) to encode graph and infer node representations. Here the implementation of GNNs is formulated as follows:

$$\begin{aligned} \bar{\mathbf{A}} &= \bar{\mathbf{D}}^{-\frac{1}{2}}(\tilde{\mathbf{A}})\bar{\mathbf{D}}^{-\frac{1}{2}} \\ \mathbf{X}_{i+1} &= \mathrm{GNN}_i(\mathbf{X}_i, \bar{\mathbf{A}}) = \bar{\mathbf{A}}\mathbf{X}_i\mathbf{W}_i \end{aligned} \tag{21}$$

where $\tilde{\mathbf{A}}$ is a self-loop adjacency matrix with $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_n$, $\bar{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$ with $\bar{\mathbf{D}} = \sum_j \tilde{\mathbf{A}}_{i,j}$, $\mathbf{X}_0$ is set default as $\mathbf{I}_n$, and $\mathbf{W}_i$ is the parameters of the i-th layer of GNNs. To handle the over-smoothing problem, we use the PairNorm [79] layer to retain the distance of node representations after each layer of GNNs. The normalization procedure is formulated as follows:

$$\begin{aligned} x_i^c &= x_i - \frac{1}{n}\sum_{i=1}^{n} x_i \\ x_i^p &= s \cdot \frac{x_i^c}{||x_i^c||^2} \end{aligned} \tag{22}$$

where x^i is the i-th node's representation, x_i^c is the node-wise centered node representation, x_i^p is the feature-wise normalized node representation, and s is the hyperparameter to adjust the distance between node representations. Our proposed encoder adds the PairNorm layer before each activation layer. Note that the first layer of GNNs requires the maximum usage of memory. For example, $\mathbf{A}$ requires $O(m)$, $\mathbf{X}_0$ requires $O(n)$, and $\mathbf{W}_0 \in \mathbb{R}^{n\times d}$ requires $O(n \times d)$, where d is the dimension of node representations. Therefore, the time and space complexity outcomes of the encoder are $O(n+m)$ and $O(m+n\times d)$, respectively.

Decoder In this paper, we choose to decode one subgraph with n_s nodes per epoch when training our proposed GAE. After obtaining the node representations from the encoder per epoch, we choose n_s nodes as our temporary ground truth subgraph $\mathbf{A}_s$. The corresponding output of the decoder is calculated as follows:

$$\begin{aligned} P(\mathbf{A}_{i,j} = 1) &= \sigma(\mathbf{Z}_i\mathbf{Z}_j^T) \\ P(\hat{\mathbf{A}}) &= \prod_{i,j\in V_s} P(\mathbf{A}_{i,j} = 1) \end{aligned} \tag{23}$$

where σ denotes the Sigmoid activation function, $\hat{\mathbf{A}}$ is the estimated adjacency matrix of the subgraph with n_s nodes, and V_s denotes the set of nodes in the subgraph.

As mentioned by Guillaume Salha et al. [55], nodes with high degrees need to be trained more frequently to avoid losing important node information. Thus, we choose subgraphs with a strategy according to node degrees as follows: $P_i = \frac{deg_i}{\sum_{i=1}^{n} deg_i}$, where P_i is the probability to select node i. In every epoch, we randomly sample nodes to assemble one subgraph for the training purpose. Note that n_s is an important hyperparameter to make a trade-off between efficiency and effectiveness. However, generating the whole graph after the entire training process still requires $O(n^2)$ time and space complexity. To address this issue, we follow the implementation of assembling adjacency matrix in NetGAN [10] and improve it as follows: (i) We obtain the latent variables of n nodes; (ii) we decode one row of the adjacency matrix through sampling edges for each node, and then clear the memory of zero entry; (iii) we repeat the step (ii) until all rows are generated. This procedure requires $O(m+n)$ space complexity, which is affordable in generating large graphs.

Optimization Since the loss calculated by each epoch is biased from the real loss $\mathcal{L}$, we use an approximate loss function $\mathcal{L}_{n_s}$ to optimize the model parameters. After each sampling, the connection status of the to-be-simulated subgraph has changed. Therefore, we dynamically sample the negative edges based on the subgraph to speed up the training process. Now the approximate loss function is formulated as follows:

$$\mathcal{L}_{n_s} = \frac{1}{m_s}[\sum_{(i,j)\in E_{pos}} (1-\hat{\mathbf{A}}_{i,j}) \log \hat{\mathbf{A}}_{i,j} + \sum_{(i,j)\in E_{neg}} \hat{\mathbf{A}}_{i,j} \log(1-\hat{\mathbf{A}}_{i,j})] \tag{24}$$

where m_s denotes the number of edges in the subgraph, $\mathbf{A}_s$, E_{pos}, and E_{neg} are the sets of positive and sampled negative edges from $\mathbf{A}_s$, respectively, and $\hat{\mathbf{A}}_{i,j}$ is the estimated probability derived from the decoder.

Overall, our proposed graph generator is a autoencoder based and we give it the name, Scalable Graph Autoencoder (**SGAE**). SGAE inherits the excellent expression performance of the graph generator based on the neural network and meanwhile achieves a significant speedup in the training process compare to other deep neural network-based generators. Particularly, the training time complexity is $O(n+m)$ and $O(n_s^2)$ in each epoch for encoder and decoder, respectively, where n_s is the size of the sampled subgraph during the training process. The inference time complexity remains $O(n^2)$, and the space complexity of SGAE is $O(m+n+n_s^2)$. Like other autoencoder-based generators, SGAE has the permutation-invariance and community preserving properties.

5 Evaluation

We integrated all included models and conducted extensive experiments to evaluate the performance of graph generators. The details of our evaluation platform are provided in Sect. 5.1. The experiment setup is provided in Sect. 5.2. Experiments were conducted in graph simulation quality, preserving community structure, parameter sensitivity, and model scalability in Sect. 5.3, Sect. 5.4, Section 5.5, and Sect. 5.6, respectively. Section 5.6 investigates the efficiency and the scalability of the graph generators. Finally, according to our comprehensive experimental study, a roadmap of recommendations is provided in Sect. 5.7 for users.

5.1 Toolkit Used for Performance Evaluation.

To help researchers and practitioners apply the general graph generators in their applications or make a comprehensive evaluation of their proposed general graph generators, we implement an end-to-end platform that is now publicly available[1]. In the detailed instruction of this toolkit, we show: (i) how to apply an existing general graph generator in user's application; (ii) how to include a new, developed general graph generator; and (iii) the details of the evaluation metrics and how to include them for performance evaluation. Currently, a Python interface is provided in our package and other programming languages will be considered in the future.

Below we briefly introduce the characteristics of the platform.

Modularized Pipelines We specify the data type as the list of Graph objects implemented under the NetworkX library [25]. The graph generators and evaluation metrics are implemented with the same type of input and output. The experimental result can be obtained directly by one-line command including a dataset, a graph generator, and a specific evaluation metric.

Customization and extension We allow users to incorporate their own datasets, graph generators, and new evaluation metrics into their local libraries by integrating their own implementations into respective source code scripts. We also welcome other developers to contribute to our platform on Github.

Note that all experiments in this paper are conducted on this platform.

[1] https://github.com/xiangsheng1325/GraphGenerator

5.2 Experiment setup

We introduce the experimental datasets, metrics, and parameter settings in this subsection.

Dataset We have collected several representative graph datasets used by existing general graph generators, which are shown in Table 3. Details of each dataset are provided as follows.

- **Citation Networks** are undirected graphs that consist of papers and their citation relationships. The Cora and Cora-ML datasets contain 2708 and 2810 machine learning publications, respectively. The Citeseer and PubMed datasets contain 3327 and 19717 publications, respectively.
- **Biological Networks** are graph-structure data extracted by real biological information. Protein dataset contains 620 nodes, each node denoting an amino acid. There are edges between amino acids when their distances are less than 6 Angstroms. Protein–protein Interaction (PPI) dataset contains 2361 nodes, each node representing one yeast protein. Edges are generated if there are interactions between two proteins.
- **Social Networks** are graph-structure data extracted by real social relationships. Deezer dataset contains 47538 nodes and each node signifies a user. The edges designate the friendship among users. The Facebook dataset contains 50515 nodes, each node denoting one page. Edges are generated if there are mutual likes among them.
- **Other Datasets** are graph-structure data from real objects. 3D point cloud dataset contains 5037 nodes, denoting the points of a household object. Edges are generated for k-nearest neighbors which are measured w.r.t Euclidean distance of the points in 3D space. Autonomous system dataset contains 6474 nodes, which represent routers of computer networks. Edges denote the communication among routers.

Note that, for some observed data with isolated nodes or self-loop edges, experiments on Recurrent Neural Network (RNN)-based graph generators (e.g., GraphRNN [77] and BiGG [18]) and NetGAN cannot be conducted successfully. Therefore, the self-loop edges of all datasets are removed. The largest connected component is selected as the input of the graph generative model.

5.2.1 Evaluating metrics

We collected and designed appropriate evaluating metrics to measure the difference between the original graph and the generated graph. The metrics used for graph simulation quality can be categorized into the following four aspects.

- **Node Distributions** are measured by using the maximum mean discrepancy (MMD) over *Degree*, *Clustering Coefficient*, *Spectral Embedding*, *Betweenness Centrality*, and *Closeness Centrality* distributions. The squared MMD between two sets of samples from distributions p and q can be derived as:

$$\begin{aligned}\mathrm{MMD}^2(p||q) = \ & \mathbb{E}_{x,y\sim p}[k(x,y)] + \mathbb{E}_{x,y\sim q}[k(x,y)] \\ & - 2\mathbb{E}_{x\sim p,y\sim q}[k(x,y)].\end{aligned} \tag{25}$$

  where k denotes the associated kernel. We use the earth mover's distance (EMD) as the Gaussian kernels, which is formulated as:

$$EMD(p,q) = \inf_{\gamma\in\prod(p,q)} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||] \tag{26}$$

  where $\prod(p,q)$ denote the set of all distributions whose marginals are p and q, respectively, and γ is a transport plan.
- **Graphlet Distributions** are measured through computing the number of occurrences of all graphlets within 4 nodes and using *Orbit* MMD to formulate the distance between two distributions with a Gaussian kernel of the total variation (TV), which is formulated as:

$$\mathrm{TV}(p,q) = \mathbb{E}_i[||\pi_p(i) - \pi_q(i)||] \tag{27}$$

  where $\pi_p(i)$ denotes probability of the i-th graphlet in graph distribution p.
- **Graph Statistics** are measured by 3 metrics: *Characteristic path length* (CPL), *Gini Coefficient* (GINI), and *Power-law Exponent* (PLE). CPL denotes the average value of the minimum path length of total node pairs. GINI denotes the inequality in the nodes' degree distribution. PLE is the exponent of the power-law distribution. All metrics reported in the experiments represent the distances between generated graphs and observed graphs.
- **Community Structures** are measured in two steps: modeling community structure and compare the differences between community ownerships of nodes. For one generated/observed graph, we first use the louvain [9] community detection algorithm to obtain its community memberships of nodes. Then we leverage *Normalized Mutual Information* (NMI) and *Adjusted Rand Index* (ARI) to measure the similarity of the community structure between two graphs.

Note that for all these graph simulation quality-related metrics, the smaller value is preferred in the performance evaluation. For all MMD-based evaluation metrics, the standard deviation of *Orbit* is set to 30, and the other standard

Table 3 Benchmark datasets included in the experiments

Category	Dataset	#Nodes	#Edges
Citation Networks	Cora [58]	2708	5429
	Citeseer [58]	3327	4732
	PubMed [58]	19717	44338
	Cora-ML [45]	2810	7981
Biological Networks	Protein [19]	620	1098
	PPI [13]	2361	6646
Social Networks	Deezer [54]	47538	222887
	Facebook [54]	50515	819090
Other Datasets	3D Point Cloud [48]	5037	10886
	Autonomous System [38]	6474	12572
	EU Email [38]	265214	364481
	Google Pages [39]	875713	4322051

deviations are set to 1.0. We use the implementation of *Orbit* counting in [77] to calculate 4-node graphlets to improve efficiency. We repeat sampling 200 nodes from the observed graph to estimate the betweenness centrality of each node. Other settings of evaluating metrics are the same as [18] and [10].

Apart from the graph simulation quality-related metrics, we also evaluate the performance of the graph generators from many other perspectives including training time, inference time (i.e., the time used for generating a new graph), scalability, space (i.e., memory consumption), robustness, community preserving, stableness, and sensitivity.

5.2.2 Parameter Settings

This section introduces the configuration and parameter settings. By default, we use the best parameter setting given by the original authors. The graph generators and evaluating scripts are implemented and compiled through Python-3.6, PyTorch-1.8.1, CUDA-11.1, and GCC-4.8.5 in our experiments. The experiments are operated on a machine with Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 80 GB RAM, and NVIDIA RTX 3090 with 24 GB memory. We use one CPU core and one GPU for every algorithm.

The initiator matrix $\mathbf{M}_I$ of R-MAT is $\{0.90.3, 0.30.1\}$ by default. Following [18,42,77], the *maximum previous number of nodes* in RNN-based graph generators (e.g., GraphRNN, GRAN, and BiGG) is set as the number of nodes in the observed graph. The stride of GRAN is 1. For all autoencoder-based graph generators and ARVGA, the node attributes $\mathbf{X}$ of each graph are configured based on the matrix $\mathbf{I}_n$, which means each node is represented by a one-hot vector. Moreover, in the training process of GAEs and ARVGA, 20% of observed edges are being masked. For assembling a graph in NetGAN, the number of random walks sampled from the trained model is 1000. We leverage spectral embeddings as the input node features of CondGEN.

5.3 Graph simulation quality

This section evaluates the quality of simulated graphs via a set of evaluation metrics. For each generator, we report the average result value of 10 repeated experiments.

First, we report the experimental results for all graph simulation quality-related metrics. Table 4 summarizes all graph generators' performances on the protein dataset. According to the 14th and 15th rows of Table 4, we can see that GraphRNN and its variant GraphRNN-S outperform other graph generators. Apart from GraphRNN and GraphRNN-S, GRAN and BiGG also perform well for the protein dataset. It can be seen that the quality of graphs GraphRNN generated is the best. Other graph generators cannot achieve the best performance in all metrics as well. Taking W-S as an example, according to Table 5, W-S has achieved the best performance in the first column, i.e., degree metric. However, W-S performs poorly in other evaluating metrics. Comparing all graph generators and all evaluating metrics in Table 5 we come to the finding that no graph generator can beat other graph generators on all evaluating metrics in the Autonomous Systems dataset. This is not restricted to this dataset; we also find that in six datasets (i.e., Cora, Citeseer, Cora-ML, PPI, 3D Point Cloud, Autonomous Systems), no graph generator beats others in all evaluating metrics. Due to space constraints, those experimental results which indicate similar findings are omitted in the text.

We also report the results of several representative evaluating metrics on all datasets. We provide the quality details of each generator for *Degree*, *Clustering Coefficient*, and *Orbit* in Table 6, Table 7, and Table 8, respectively. We can see that SGAE achieves seven out of twelve best results in *Degree* from Table 6, five out of twelve best results in *Clustering*

Table 4 Evaluation results on protein dataset

Graph	Generator	Degree.	Cluster.	Orbit	Spec.	Between.	Close.	Charact. Path Len.	Gini Coeffi.	Power-law Expo.
E-R		$5.01e^{-2}$	1.92	$6.23e^{-2}$	0.142	0.755	$3.42e^{-2}$	17.5	$8.86e^{-2}$	0.12
W-S		0.139	1.96	0.97	0.203	0.115	0.144	5.25	$2.46e^{-2}$	0.842
B-A		$5.54e^{-2}$	1.77	1.18	0.327	0.754	$7.83e^{-2}$	19.4	0.159	1.43
RTG		$9.71e^{-2}$	1.29	0.282	0.351	0.781	$9.40e^{-2}$	17.8	$9.89e^{-2}$	$3.28e^{-2}$
BTER		$1.91e^{-2}$	1.01	0.270	0.263	0.234	$6.12e^{-2}$	10.2	$9.63e^{-2}$	$3.01e^{-2}$
SBM		$5.38e^{-2}$	1.18	0.102	0.175	0.733	$3.20e^{-2}$	12.4	$4.87e^{-2}$	$5.13e^{-2}$
DCSBM		$9.43e^{-2}$	0.89	0.366	0.215	0.749	$4.27e^{-2}$	13.3	0.142	$8.14e^{-2}$
R-MAT		0.258	0.98	1.18	0.323	0.707	0.131	19.7	0.335	0.258
Kronecker		0.101	1.95	1.96	0.268	0.771	$5.74e^{-2}$	18.5	0.132	$3.12e^{-2}$
MMSB		0.116	1.94	0.326	0.199	0.758	$6.25e^{-2}$	17.9	0.173	0.186
VGAE		0.447	1.56	1.87	0.6	0.535	0.351	18.2	0.477	0.126
Graphite		0.498	2	2	0.629	0.591	0.422	20.1	0.473	0.134
SBMGNN		0.513	1.6	1.94	0.65	0.668	0.374	20.5	0.52	0.209
GraphRNN		$2.32e^{-2}$	**0.407**	$6.77e^{-2}$	$5.74e^{-2}$	0.159	$2.18e^{-2}$	4.85	**1.16**e^{-2}	**2.12**e^{-2}
GraphRNN-S		**1.08**e^{-2}	0.443	**2.92**e^{-3}	**5.61**e^{-2}	**3.68**e^{-2}	**1.87**e^{-2}	**1.26**	$2.45e^{-2}$	$4.41e^{-2}$
GRAN		$4.33e^{-2}$	1.2	0.734	0.183	0.722	$3.13e^{-2}$	18.3	$1.71e^{-2}$	0.631
BiGG		$5.31e^{-2}$	1.87	$7.99e^{-2}$	0.119	0.737	$3.21e^{-2}$	17.3	$7.32e^{-2}$	0.176
ARVGA		0.465	1.33	1.28	0.42	0.766	0.263	19.5	0.501	0.105
NetGAN		$3.77e^{-2}$	1.51	0.128	0.136	0.774	$3.30e^{-2}$	16.8	$6.37e^{-2}$	$5.42e^{-2}$
CondGEN		0.312	1.15	1.1	0.442	0.547	0.376	20.8	0.362	0.295
SGAE		$3.54e^{-2}$	1.62	0.23	0.585	0.606	0.348	19.8	0.298	0.469

Table 5 Evaluation results on autonomous system dataset

Graph	Generator	Degree.	Cluster.	Orbit	Spec.	Between.	Close.	Charact. Path Len.	Gini Coeffi.	Power-law Expo.
E-R		$6.75e^{-2}$	0.127	2	$6.57e^{-2}$	0.778	0.328	8.03	0.222	$\mathbf{2.44}e^{-2}$
W-S		$\mathbf{8.54}e^{-3}$	0.127	2	$7.37e^{-2}$	0.697	0.277	97.2	0.344	0.284
B-A		$8.81e^{-2}$	0.127	1.17	$2.33e^{-3}$	0.63	0.142	5.16	0.187	1.22
RTG		$6.89e^{-2}$	$9.45e^{-2}$	2	$9.78e^{-2}$	0.901	0.269	5.05	0.27	1.13
BTER		0.310	$4.78e^{-2}$	2	$7.32e^{-2}$	0.814	0.215	3.76	0.13	0.967
SBM		0.264	0.105	2	$6.11e^{-2}$	0.778	0.387	3.25	0.31	0.651
DCSBM		$7.89e^{-2}$	$3.51e^{-2}$	1.32	$\mathbf{9.49}e^{-5}$	0.536	0.184	0.477	$3.89e^{-2}$	0.271
R-MAT		$5.16e^{-2}$	$6.17e^{-2}$	1.21	$3.53e^{-3}$	**0.102**	0.153	$\mathbf{2.83}e^{-2}$	$5.46e^{-2}$	0.498
Kronecker		0.126	$7.77e^{-2}$	1.08	$1.44e^{-3}$	0.166	0.175	0.251	$\mathbf{3.75}e^{-2}$	0.743
MMSB		$6.26e^{-2}$	$5.40e^{-2}$	1.97	$1.84e^{-4}$	0.361	$\mathbf{5.51}e^{-2}$	$6.50e^{-2}$	$8.01e^{-2}$	0.145
VGAE		$8.04e^{-2}$	$2.60e^{-2}$	1.59	$5.22e^{-4}$	0.41	$9.54e^{-2}$	0.453	0.184	$3.37e^{-2}$
Graphite		0.121	$\mathbf{1.92}e^{-2}$	2	$7.31e^{-4}$	0.462	0.123	0.746	0.218	$7.33e^{-2}$
GraphRNN		–	–	–	–	–	–	–	–	–
GraphRNN-S		–	–	–	–	–	–	–	–	–
GRAN		$7.23e^{-2}$	$8.58e^{-2}$	**0.99**	$2.67e^{-3}$	0.155	0.13	0.364	$3.83e^{-2}$	0.5
BiGG		0.109	0.119	1.89	$5.33e^{-2}$	0.769	0.343	3.27	0.25	0.461
ARVGA		0.201	$3.31e^{-2}$	1.12	$9.86e^{-4}$	0.412	0.158	1.18	0.218	0.336
NetGAN		$2.72e^{-2}$	$6.89e^{-2}$	1.07	$6.32e^{-4}$	0.157	$7.66e^{-2}$	0.436	$5.97e^{-2}$	0.251
CondGEN		–	–	–	–	–	–	–	–	–
SGAE		$2.52e^{-2}$	$3.47e^{-2}$	1.01	$6.93e^{-2}$	0.296	0.46	0.912	0.273	0.845

Table 6 Evaluation results of *Degree* MMD on all 12 datasets

Graph	Generator	Cora	Citeseer	PubMed	Cora-ML	Protein	PPI	Deezer	Facebook	3D Point Cloud	Autonom. System	**EU Email**	**Google**
E-R		$3.79e^{-3}$	$1.18e^{-2}$	$4.38e^{-2}$	$1.88e^{-2}$	$1.06e^{-2}$	$4.41e^{-2}$	$3.61e^{-2}$	0.11	$9.71e^{-2}$	$1.66e^{-2}$	$6.40e^{-2}$	$6.16e^{-2}$
W-S		$3.62e^{-2}$	$1.81e^{-2}$	$2.44e^{-2}$	$5.25e^{-2}$	$3.71e^{-2}$	$3.26e^{-2}$	0.128	0.14	0.23	$\mathbf{2.09}e^{-3}$	$6.27e^{-2}$	$9.28e^{-2}$
B-A		$2.09e^{-2}$	$2.17e^{-2}$	$7.20e^{-2}$	$3.86e^{-2}$	$1.34e^{-2}$	$7.09e^{-2}$	$4.33e^{-2}$	$8.03e^{-2}$	0.151	$2.19e^{-2}$	$8.97e^{-3}$	$6.75e^{-2}$
RTG		$3.65e^{-2}$	$2.82e^{-2}$	$4.16e^{-3}$	$2.73e^{-2}$	0.141	$1.93e^{-2}$	$7.37e^{-2}$	0.101	0.232	$2.93e^{-2}$	$5.37e^{-4}$	–
BTER		$9.44e^{-3}$	$2.02e^{-3}$	$8.02e^{-3}$	$9.31e^{-3}$	$1.37e^{-2}$	$4.16e^{-3}$	$1.37e^{-4}$	$3.28e^{-3}$	$7.03e^{-2}$	$4.76e^{-3}$	$3.33e^{-2}$	$1.58e^{-3}$
SBM		$8.59e^{-3}$	$1.26e^{-2}$	$6.55e^{-2}$	$2.87e^{-2}$	$1.52e^{-2}$	$5.34e^{-2}$	$3.71e^{-2}$	$7.55e^{-2}$	$8.24e^{-2}$	$6.85e^{-2}$	0.113	$5.86e^{-2}$
DCSBM		$6.99e^{-3}$	$8.10e^{-3}$	$1.52e^{-2}$	$\mathbf{3.99}e^{-3}$	$2.58e^{-2}$	$7.83e^{-3}$	$8.34e^{-4}$	$9.27e^{-4}$	$8.18e^{-2}$	$1.95e^{-2}$	$6.29e^{-2}$	$2.44e^{-3}$
MMSB		$6.79e^{-3}$	$8.27e^{-3}$	–	$5.58e^{-3}$	$2.85e^{-2}$	$8.11e^{-3}$	–	–	$9.36e^{-2}$	$1.52e^{-2}$	–	–
R-MAT		$2.62e^{-2}$	$1.42e^{-2}$	$2.84e^{-3}$	$1.71e^{-2}$	$6.57e^{-2}$	$2.96e^{-3}$	$4.12e^{-2}$	$2.25e^{-2}$	0.175	$1.31e^{-2}$	$3.85e^{-2}$	$1.51e^{-2}$
Kronecker		$1.32e^{-2}$	$1.24e^{-2}$	$1.17e^{-2}$	$4.49e^{-3}$	$2.38e^{-2}$	$4.00e^{-3}$	$2.93e^{-3}$	$1.34e^{-2}$	0.102	$3.40e^{-2}$	$6.35e^{-2}$	$6.67e^{-3}$
VGAE		$5.11e^{-2}$	$8.65e^{-2}$	0.158	$5.14e^{-2}$	0.115	$5.71e^{-2}$	–	–	0.213	$2.15e^{-2}$	–	–
Graphite		$5.44e^{-2}$	$7.58e^{-2}$	0.176	$5.02e^{-2}$	0.138	0.1	–	–	0.196	$3.11e^{-2}$	–	–
SBMGNN		$8.10e^{-2}$	0.112	0.128	$6.80e^{-2}$	0.144	0.107	–	–	0.161	$3.88e^{-2}$	–	–
GraphRNN		–	–	–	–	$2.42e^{-3}$	–	–	–	–	–	–	–
GraphRNN-S		$7.69e^{-3}$	$2.56e^{-2}$	–	$2.55e^{-2}$	$\mathbf{1.52}e^{-3}$	$4.13e^{-2}$	–	–	–	–	–	–
GRAN		$7.06e^{-3}$	$2.06e^{-2}$	–	$1.28e^{-2}$	$1.34e^{-2}$	$3.37e^{-2}$	–	–	0.145	$2.00e^{-2}$	–	–
BiGG		$3.05e^{-3}$	$\mathbf{1.09}e^{-3}$	–	$8.58e^{-3}$	$1.58e^{-2}$	$3.01e^{-2}$	–	–	0.105	$2.83e^{-2}$	–	–
ARVGA		$7.92e^{-2}$	0.114	0.17	$2.40e^{-2}$	0.122	$5.12e^{-2}$	–	–	0.209	$5.24e^{-2}$	–	–
NetGAN		$\mathbf{1.28}e^{-3}$	$1.61e^{-3}$	–	$4.71e^{-3}$	$9.04e^{-3}$	$1.02e^{-2}$	–	–	$6.89e^{-2}$	$6.43e^{-3}$	–	–
CondGEN		$4.69e^{-2}$	$1.40e^{-2}$	–	$2.94e^{-2}$	$8.61e^{-2}$	$1.22e^{-2}$	–	–	0.175	–	–	–
SGAE		$6.37e^{-3}$	$2.91e^{-3}$	$\mathbf{2.06}e^{-3}$	$5.29e^{-3}$	$2.92e^{-3}$	$\mathbf{1.84}e^{-3}$	$\mathbf{1.30}e^{-4}$	$\mathbf{8.69}e^{-4}$	$\mathbf{6.12}e^{-2}$	$2.90e^{-3}$	$\mathbf{1.97}e^{-4}$	$\mathbf{9.38}e^{-4}$

Table 7 Evaluation results of *Clustering Coefficient* MMD on all 12 datasets

Graph	Generator	Cora	Citeseer	PubMed	Cora-ML	Protein	PPI	Deezer	Facebook	3D Point Cloud	Autonom. System	EU Email	Google
E-R		$7.35e^{-2}$	$3.90e^{-2}$	$1.46e^{-2}$	0.101	$1.72e^{-2}$	$4.01e^{-2}$	0.117	0.146	$5.97e^{-2}$	$3.29e^{-2}$	$2.19e^{-3}$	0.132
W-S		$7.64e^{-2}$	$4.03e^{-2}$	$1.46e^{-2}$	0.104	$1.98e^{-2}$	$4.22e^{-2}$	0.12	0.151	$6.05e^{-2}$	$3.31e^{-2}$	$2.19e^{-3}$	0.131
B-A		$6.22e^{-2}$	$3.36e^{-2}$	$1.37e^{-2}$	$8.18e^{-2}$	$1.38e^{-2}$	$1.96e^{-2}$	0.106	0.124	$5.80e^{-2}$	$3.31e^{-2}$	$2.19e^{-3}$	0.13
RTG		$3.16e^{-2}$	$4.24e^{-2}$	$2.93e^{-2}$	$2.83e^{-2}$	$2.23e^{-2}$	$2.11e^{-2}$	$8.02e^{-2}$	0.105	0.118	$9.64e^{-3}$	$7.20e^{-4}$	–
BTER		$\mathbf{1.37}e^{-3}$	$3.15e^{-3}$	$3.89e^{-3}$	$3.43e^{-3}$	$2.51e^{-3}$	$1.63e^{-2}$	$2.84e^{-3}$	$1.99e^{-2}$	$\mathbf{1.18}e^{-2}$	$8.40e^{-3}$	$4.24e^{-3}$	$1.99e^{-2}$
SBM		$4.90e^{-2}$	$2.14e^{-2}$	$1.05e^{-2}$	$7.48e^{-2}$	$6.11e^{-3}$	$1.84e^{-2}$	$9.95e^{-2}$	0.13	$4.57e^{-2}$	$2.79e^{-2}$	$1.99e^{-3}$	0.12
DCSBM		$2.82e^{-2}$	$1.10e^{-2}$	$3.65e^{-3}$	$4.02e^{-2}$	$4.26e^{-3}$	$7.79e^{-3}$	$9.34e^{-2}$	$3.98e^{-2}$	$4.66e^{-2}$	$9.43e^{-3}$	$2.73e^{-4}$	$8.92e^{-2}$
MMSB		$6.44e^{-2}$	$3.29e^{-2}$	–	$7.99e^{-2}$	$1.76e^{-2}$	$1.96e^{-2}$	–	–	$5.96e^{-2}$	$1.50e^{-2}$	–	–
R-MAT		$4.12e^{-2}$	$1.78e^{-2}$	$2.02e^{-3}$	$4.25e^{-2}$	$4.33e^{-3}$	$8.53e^{-3}$	$3.19e^{-2}$	$1.89e^{-2}$	$4.58e^{-2}$	$1.58e^{-2}$	$1.52e^{-3}$	0.106
Kronecker		$6.94e^{-2}$	$3.80e^{-2}$	$1.08e^{-2}$	$8.36e^{-2}$	$1.92e^{-2}$	$1.51e^{-2}$	0.113	$9.46e^{-2}$	$6.04e^{-2}$	$2.02e^{-2}$	$2.03e^{-3}$	0.13
VGAE		$3.91e^{-2}$	$2.00e^{-2}$	$1.06e^{-2}$	$5.77e^{-2}$	$4.12e^{-2}$	$1.43e^{-2}$	–	–	$5.43e^{-2}$	$7.07e^{-3}$	–	–
Graphite		$4.14e^{-2}$	$1.92e^{-2}$	$1.01e^{-2}$	$5.83e^{-2}$	$3.96e^{-2}$	$2.05e^{-2}$	–	–	$4.83e^{-2}$	$\mathbf{5.21}e^{-3}$	–	–
SBMGNN		$4.36e^{-2}$	$2.05e^{-2}$	$1.00e^{-2}$	$6.03e^{-2}$	$2.90e^{-2}$	$2.09e^{-2}$	–	–	$4.70e^{-2}$	$5.62e^{-3}$	–	–
GraphRNN		–	–	–	–	$\mathbf{1.33}e^{-3}$	–	–	–	–	–	–	–
GraphRNN-S		$2.52e^{-2}$	$1.33e^{-2}$	–	$7.52e^{-2}$	$3.20e^{-3}$	$3.12e^{-2}$	–	–	–	–	–	–
GRAN		$6.66e^{-2}$	$1.49e^{-2}$	–	$6.19e^{-2}$	$7.57e^{-3}$	$2.50e^{-2}$	–	–	0.102	$2.25e^{-2}$	–	–
BiGG		$7.15e^{-2}$	$3.79e^{-2}$	–	$9.32e^{-2}$	$1.45e^{-2}$	$\mathbf{3.13}e^{-3}$	–	–	$4.96e^{-2}$	$3.14e^{-2}$	–	–
ARVGA		$2.36e^{-2}$	$1.34e^{-2}$	$7.19e^{-3}$	$\mathbf{1.84}e^{-2}$	$1.22e^{-2}$	$5.90e^{-3}$	–	–	$4.03e^{-2}$	$8.47e^{-3}$	–	–
NetGAN		$1.08e^{-2}$	$\mathbf{2.76}e^{-3}$	–	$5.00e^{-2}$	$6.62e^{-3}$	$1.87e^{-2}$	–	–	$1.71e^{-2}$	$1.88e^{-2}$	–	–
CondGEN		0.106	0.104	–	$9.83e^{-2}$	0.165	$9.92e^{-2}$	–	–	0.196	–	–	–
SGAE		$1.99e^{-2}$	$2.73e^{-2}$	$\mathbf{1.79}e^{-3}$	$2.08e^{-2}$	$1.83e^{-2}$	$1.03e^{-2}$	$\mathbf{1.61}e^{-3}$	$\mathbf{1.31}e^{-2}$	$3.64e^{-2}$	$1.54e^{-2}$	$\mathbf{1.70}e^{-4}$	$\mathbf{5.94}e^{-3}$

Table 8 Evaluation results of *Orbit* MMD on all 12 datasets

Graph	Generator	Cora	Citeseer	PubMed	Cora-ML	Protein	PPI	Deezer	Facebook	3D Point Cloud	Autonom. System	EU Email	**Google**
E-R		0.204	$3.83e^{-2}$	$2.08e^{-2}$	0.999	$2.69e^{-4}$	0.564	$3.28e^{-4}$	2	$3.65e^{-5}$	2	2	0.77
W-S		0.616	0.107	$3.26e^{-2}$	1.91	$1.20e^{-2}$	1.51	$1.29e^{-2}$	2	$3.63e^{-4}$	2	2	0.968
B-A		0.177	$9.98e^{-2}$	$2.72e^{-3}$	0.105	0.349	$6.76e^{-2}$	$4.73e^{-2}$	2	0.134	2	2	0.254
RTG		2	2	2	2	2	2	2	2	2	2	2	–
BTER		$5.38e^{-2}$	$\mathbf{2.92}e^{-3}$	$1.89e^{-3}$	0.127	$2.89e^{-3}$	$2.77e^{-2}$	$\mathbf{2.45}e^{-5}$	0.481	$5.60e^{-5}$	2	2	0.138
SBM		0.146	$2.76e^{-2}$	$5.35e^{-3}$	0.565	$9.42e^{-4}$	0.207	$1.90e^{-4}$	1.99	$1.64e^{-4}$	2	2	0.435
DCSBM		$4.80e^{-2}$	$5.40e^{-3}$	$2.65e^{-4}$	0.231	$2.14e^{-3}$	$\mathbf{9.85}e^{-3}$	$5.83e^{-5}$	0.562	$1.76e^{-4}$	2	2	0.269
MMSB		$2.18e^{-2}$	$7.20e^{-3}$	–	$9.49e^{-2}$	$9.65e^{-4}$	$3.31e^{-2}$	–	–	$1.01e^{-4}$	2	–	–
R-MAT		1.83	1.75	1.91	2	1.92	1.99	2	2	1.99	2	2	2
Kronecker		$2.39e^{-2}$	$3.48e^{-3}$	$2.75e^{-2}$	0.322	$1.19e^{-2}$	1.49	$2.89e^{-3}$	2	$3.32e^{-4}$	2	1.91	0.141
VGAE		1.78	0.984	2	1.95	$6.31e^{-2}$	1.07	–	–	1.76	1.99	–	–
Graphite		1.97	0.979	1.62	1.98	$5.36e^{-2}$	1.61	–	–	$8.85e^{-2}$	2	–	–
SBMGNN		1.75	0.784	2	1.99	$5.30e^{-2}$	1.64	–	–	$2.10e^{-3}$	2	–	–
GraphRNN		–	–	–	–	$4.08e^{-5}$	–	–	–	–	–	–	–
GraphRNN-S		0.194	$6.70e^{-3}$	–	1.9	$\mathbf{1.51}e^{-5}$	0.237	–	–	–	–	–	–
GRAN		$6.57e^{-2}$	0.191	–	1.93	$2.51e^{-3}$	0.192	–	–	2	2	–	–
BiGG		0.181	$5.09e^{-2}$	–	0.802	$2.49e^{-4}$	0.421	–	–	$2.29e^{-4}$	2	–	–
ARVGA		1.02	0.968	2	2	0.559	1.47	–	–	0.929	2	–	–
NetGAN		$4.45e^{-2}$	$6.70e^{-3}$	–	$2.16e^{-2}$	$6.00e^{-4}$	$2.36e^{-2}$	–	–	$\mathbf{3.28}e^{-5}$	2	–	–
CondGEN		2	2	–	2	2	2	–	–	2	–	–	–
SGAE		$\mathbf{1.49}e^{-3}$	$3.31e^{-3}$	$\mathbf{2.57}e^{-4}$	$\mathbf{1.36}e^{-2}$	$1.65e^{-3}$	$9.69e^{-2}$	$3.02e^{-2}$	**0.37**	$3.35e^{-5}$	**0.577**	**1.78**	**0.107**

Coefficient from Table 7, and seven best results in *Orbit* from Table 8. Therefore, SGAE outperforms other generators, especially when generating large datasets (e.g., Google, Deezer, EU email, Facebook). SGAE cannot perform well when generating certain small-sized graphs (e.g., Protein, PPI, Cora, Citeseer). Neural network-based graph generators (e.g., GraphRNN, NetGAN, SBMGNN, etc) outperform SGAE in such datasets, but they cannot simulate large data distributions due to the limit of memory and FLOPS.

In terms of specific properties, Table 7 shows that BTER generates graphs with better clustering coefficient distribution than most generators. That is because BTER generates edges directly referring to the clustering coefficient by each node. Although SGAE outperforms BTER, the latter is much faster. The detailed speed comparison is introduced in Sect. 5.6.

In general, although there are several generators (e.g., GraphRNN) ranking higher SGAE when generating small graphs, these results still prove that SGAE fills the gap in learning the generative distribution of large networks. That is because SGAE breaks the scalability limitation of neural network-based graph generators.

5.4 Preserving community structure

In this subsection, we compare the performance of graph generators in terms of preserving community structures. We report experimental results on four representative datasets (Cora, Cora-ML, PPI, 3D Point Cloud) in Table 9. Some graph generators (e.g., E-R, RTG, and GRAN) are excluded because their optimization objectives cannot preserve node order and community membership.

We find that the autoencoder-based graph generators (i.e., VGAE, Graphite, SBMGNN, and SGAE) are superior in preserving community structures in Cora, Cora-ML, and PPI datasets. NetGAN performs well on 3D Point Cloud. This is because the autoencoder-based graph generators leverage the graph neural network (GNN) to infer the node representations, which will be decoded into a new graph. The GNN architecture has a reliable reasoning capability for community memberships and better generalization performance than NetGAN. In certain cases, users may prefer preserving community memberships, i.e., community structures of the observed graph. The autoencoder-based graph generator is the best choice to generate new graphs with similar community structures.

5.5 Parameter sensitivity

Since different types of graph generators have different hyperparameters, we compare them in groups according to their similarity of model architecture and test their robustness and training difficulty.

5.5.1 Sensitivity

This section evaluates the hyperparameter sensitivity of each neural network-based model and reports their performance by varying parameter settings. In Fig. 4, due to space limitation, we provide detailed comparison results for two representative evaluating metrics (Degree and Gini Index) on Protein. According to the implementation of the model, neural networks-based graph generators are divided into three categories, namely RNN-based, autoencoder-based, and GAN-based. Figure 4 shows experimental results of graph generators in these categories.

In the left part of Fig. 4, we find that, for RNN-based graph generators, GraphRNN-S outperforms others in terms of sensitivity and performance. The full GraphRNN also has a flat curve, showing robustness as good as GraphRNN-S. GRAN and BiGG occasionally generate bad results which show their high sensitivity on parameter settings. The middle section of Fig. 4 shows that VGAE generates graphs with the best robustness and quality. This is because VGAE

Table 9 Evaluation results on preserving community experiment

Methods	Cora		Cora-ML		PPI		3D Point Cloud	
	NMI(e^{-2})	ARI(e^{-2})	NMI(e^{-2})	ARI(e^{-2})	NMI(e^{-2})	ARI(e^{-2})	NMI(e^{-2})	ARI(e^{-2})
SBM	11.3 ± 0.7	1.2 ± 0.1	15.7 ± 1.1	8.9 ± 0.6	9.3 ± 0.9	1.5 ± 0.3	37.0 ± 1.3	11.4 ± 0.7
DCSBM	18.6 ± 0.8	1.8 ± 0.3	22.1 ± 0.7	7.7 ± 0.4	21.7 ± 0.7	2.5 ± 0.2	37.3 ± 1.4	11.5 ± 0.8
MMSB	15.4 ± 0.6	0.8 ± 0.4	23.3 ± 0.6	13.2 ± 1.1	22.7 ± 0.7	6.3 ± 1.0	7.1 ± 0.4	1.3 ± 0.3
VGAE	60.4 ± 0.6	40.0 ± 1.2	59.0 ± 0.9	46.4 ± 1.6	**49.1 ± 0.5**	**16.6 ± 1.8**	57.0 ± 0.8	8.2 ± 1.1
Graphite	**62.3 ± 0.8**	**43.4 ± 1.9**	60.8 ± 0.7	51.0 ± 1.2	41.1 ± 0.5	-0.2 ± 0.1	58.8 ± 0.4	13.2 ± 0.3
SBMGNN	61.9 ± 0.4	41.0 ± 1.6	**63.1 ± 0.7**	**55.0 ± 1.0**	47.8 ± 0.7	13.3 ± 0.1	59.2 ± 0.9	15.9 ± 1.1
ARVGA	41.8 ± 0.8	9.3 ± 1.3	33.1 ± 0.7	14.9 ± 1.1	32.6 ± 0.9	0.1 ± 0.2	47.2 ± 0.8	4.1 ± 0.5
NetGAN	5.2 ± 0.5	0.2 ± 0.1	37.2 ± 1.4	32.7 ± 2.6	8.5 ± 0.8	4.4 ± 0.8	**67.4 ± 0.9**	**38.8 ± 2.6**
SGAE	62.0 ± 0.7	42.2 ± 1.3	62.1 ± 0.9	54.8 ± 1.5	41.6 ± 0.9	16.0 ± 1.1	60.6 ± 0.6	35.9 ± 1.4

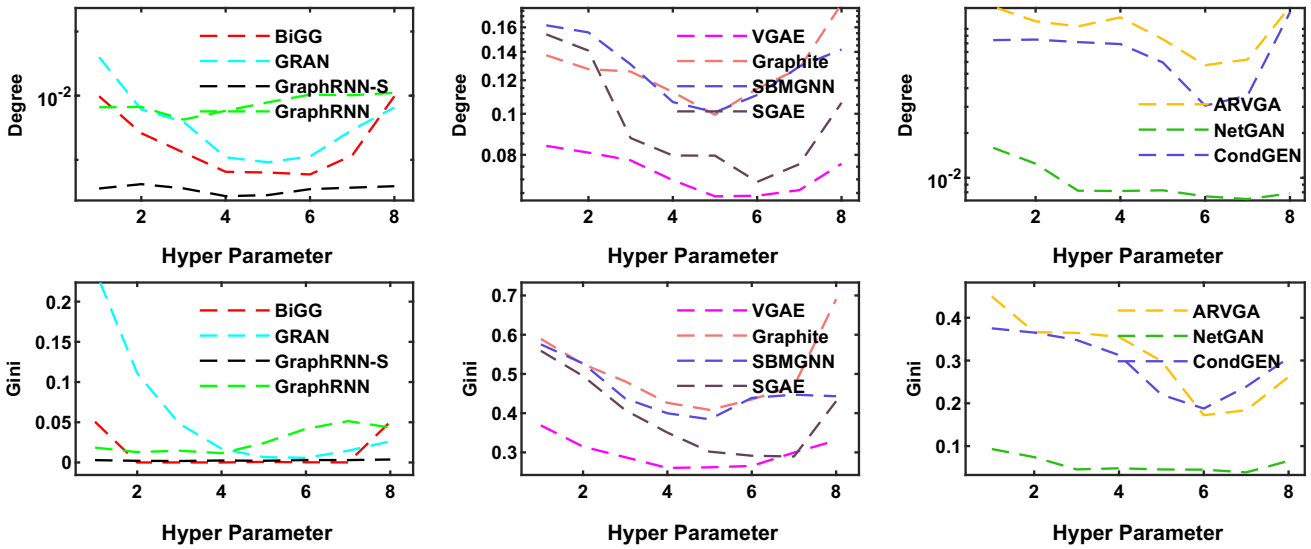

Fig. 4 Parameter sensitivity experiment results. Lower is better

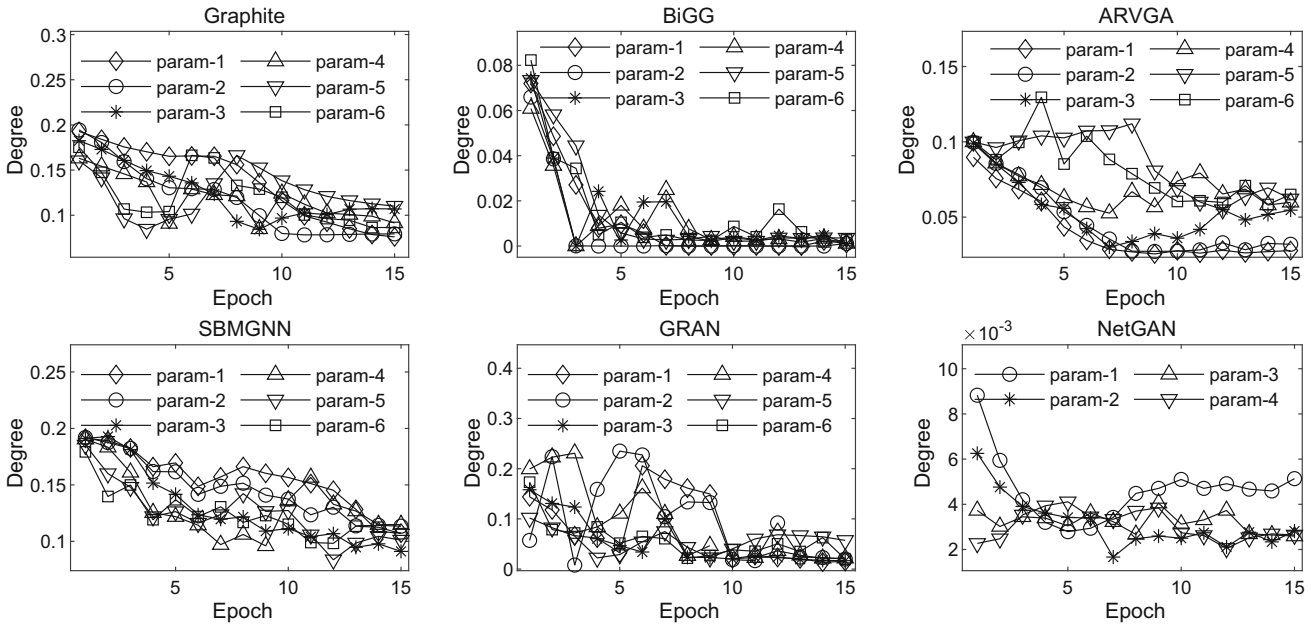

Fig. 5 Model stability experiment results

has the simplest model architecture, resulting in better generalization. On the right side of Fig. 4, NetGAN is shown clearly that it is not sensitive about parameter settings, and the quality of its generated graph is the best. The parameter changes of CondGEN and ARVGA significantly affect their generative performance compared with NetGAN. From this section, we can see that GraphRNN, VGAE, and NetGAN from three groups of graph generators are not sensitive about their parameter settings compared with the others.

5.5.2 Stability

We report the training stability of graph generators by varying parameter settings in Fig. 5. We show the experimental results on Citeseer as a representative example.

We find that the curves of the model training process of autoencoder-based graph generators (e.g., Graphite, SBMGNN) fluctuate, but the overall trend is convergent. This is because updating parameters may have an impact on the degrees of all nodes. The RNN-based graph generators (e.g.,

GRAN, BiGG) are relatively stable, and the occasional collapse does not affect its tendency to continue converging to the optimal. The graph generation occasionally cannot converge in GAN-based graph generators (e.g., ARVGA), while only the NetGAN with WGAN architecture has a more stable training process.

5.6 Scalability and efficiency

We evaluate the scalability and efficiency of each generator in this subsection. Table 10 reports the time consumption of inferring a new graph. Note that because generating graphs by NetGAN requires one more step, i.e., assembling generated random walks into an adjacency matrix, generating one graph by NetGAN is more time-consuming. Table 11 reports the time consumption of parameter updating during the training process. Table 12 chronicles the time consumption of the entire training process. Table 13 details the peak memory usage during training process.

Simple model-based graph generators have the highest efficiency for generating large networks, incurring minor extra space cost and taking little time. Combining the experimental results on Protein and PubMed in Sect. 5.3, we can see that the scalability of simple model-based graph generators (e.g., BTER, R-MAT) is better than others'. This is

Table 10 Time consumption (seconds) per graph generation

#Nodes	0.1k	1k	10k	100k	1000k
E-R	**4.6**e^{-4}	9.0e^{-3}	0.46	10.1	217
B-A	1.0e^{-3}	1.2e^{-2}	0.11	1.17	59.1
W-S	7.2e^{-4}	**7.1**e^{-3}	**0.08**	**0.81**	**8.63**
RTG	8.3e^{-3}	0.13	2.62	4.13	663
BTER	1.88e^{-3}	0.03	0.32	4.91	82.8
SBM	6.1e^{-3}	0.09	2.58	37.1	545
DCSBM	6.2e^{-3}	0.09	2.69	39.3	570
MMSB	6.1e^{-3}	0.09	2.56	–	–
R-MAT	8.5e^{-3}	0.09	0.98	9.71	99.1
Kronecker	8.5e^{-3}	0.08	1.00	9.69	99.2
GraphRNN	0.31	5.62	–	–	–
GraphRNN-S	0.27	4.74	63.6	–	–
GRAN	0.36	4.02	–	–	–
BiGG	0.33	2.03	60.4	–	–
VGAE	4.2e^{-3}	0.04	0.38	–	–
Graphite	6.1e^{-3}	0.06	0.64	–	–
SBMGNN	0.01	0.11	1.18	–	–
ARVGA	4.8e^{-3}	0.04	0.42	–	–
NetGAN	8.7e^{-3}	0.09	1.12	–	–
CondGEN	8.3e^{-3}	0.15	–	–	–
SGAE	4.5e^{-3}	0.04	0.48	43.8	4160

Table 11 Time consumption (minutes) of parameter updating during the training process

#Nodes	0.1k	1k	10k	100k	1000k
MMSB	5.4e^{-2}	0.44	18.0	–	–
Kronecker	9.6e^{-2}	0.35	**0.96**	**2.28**	**5.61**
GraphRNN	0.59	10.4	–	–	–
GraphRNN-S	0.56	6.45	61.8	–	–
GRAN	0.13	6.84	–	–	–
BiGG	0.11	2.81	33.7	–	–
VGAE	0.03	**0.10**	1.74	–	–
Graphite	0.04	0.11	2.11	–	–
SBMGNN	0.09	0.31	5.22	–	–
ARVGA	0.04	0.12	1.79	–	–
NetGAN	0.12	0.46	7.61	–	–
CondGEN	0.05	0.19	–	–	–
SGAE	0.04	0.11	1.76	8.63	79.8

Table 12 Time consumption (minutes) of the entire training process

#Nodes	0.1k	1k	10k	100k	1000k
MMSB	0.11	0.91	40.3	–	–
Kronecker	1.39	1.55	3.25	**4.73**	**21.3**
GraphRNN	2.45	28.9	–	–	–
GraphRNN-S	1.63	15.4	161	–	–
GRAN	1.36	14.3	–	–	–
BiGG	0.88	9.67	139	–	–
VGAE	**0.06**	0.42	9.75	–	–
Graphite	0.07	0.47	10.6	–	–
SBMGNN	0.08	0.63	12.4	–	–
ARVGA	0.07	0.50	10.3	–	–
NetGAN	0.27	2.80	31.1	–	–
CondGEN	0.18	25.3	–	–	–
SGAE	0.17	**0.34**	**3.15**	17.34	163.1

Table 13 Peak GPU memory usage (MiB) during training

#Nodes	0.1k	1k	10k	100k	1000k
MMSB	**1575**	**1709**	18529	OOM	OOM
GraphRNN	1915	3121	OOM	OOM	OOM
GraphRNN-S	1913	1959	5501	OOM	OOM
GRAN	1959	2677	OOM	OOM	OOM
BiGG	2043	3145	18985	OOM	OOM
VGAE	1719	1759	4799	OOM	OOM
Graphite	1719	1761	4819	OOM	OOM
SBMGNN	1719	1767	5243	OOM	OOM
ARVGA	1719	1762	4832	OOM	OOM
NetGAN	2237	2552	5008	OOM	OOM
CondGEN	1722	1789	–	–	–
SGAE	1721	1740	**1943**	**3971**	**24248**

because they are designed to generate a set of random graphs having specific properties, which is insensitive to the size of the graph. The efficiency of GraphRNN and other RNN-based graph generators is the worst since when generating large graphs, RNN needs to stack too many layers. Therefore, spending a lot of space in storing long-term memory exceeds RNN-based graph generators' ability to articulate the network. SGAE achieves the greatest efficiency in the training process and memory usage, proving the value of its decoder's improvements on time and space complexity.

5.7 Recommendation

As shown in our comprehensive performance evaluation, there is no algorithm which can win under all metrics. This is because the sophisticated models are required to achieve a good simulation quality and this inevitably sacrifices the efficiency and scalability compared to the simple models. Moreover, as there are many different metrics to measure the simulation quality, it is difficult for a graph generator to win under all these metrics because: (1) we cannot directly optimize the distribution of the generated graphs according to the observed graphs; and (2) some simple algorithms only focus on optimizing a particular metric (e.g., degree distribution). Thus, it is critical to have a comprehensive recommendation for users with different requirements, and the algorithms which can achieve good trade-off among the metrics are welcomed in practice.

Table 14 quantitatively evaluates the performance of the 20 representative general graph generators from various perspectives including graph simulation quality, training time, inference time (i.e., the time used for generating a new graph), scalability, space (i.e., memory consumption), robustness, community preserving, and tuning difficulty for users. According to the comprehensive experimental evaluation in the subsections above, for each metric, we use the number of ★ to indicate the quality of every graph generator's performance, where ★★★★★ corresponds to the best performance (i.e., fastest, most scalable, uses least memory, easiest to tune parameters). Note that we use the number of parameters, stableness, and sensitivity of the models to evaluate the user-friendliness of the parameter tuning. Therefore, Table 14 provides a roadmap of recommendations for researchers and practitioners in how to select general graph generators in different settings.

Below are some recommendations for users according to our comprehensive evaluations.

- The best graph generator tool for simulating a citation network is our proposed SGAE. According to the last row of Table 8, SGAE achieves three best performances for the four citation-network datasets (Cora, Citeseer, PubMed, Cora-ML). Furthermore, according to Tables 6 and 7 , SGAE also achieves the best performance for the PubMed dataset. The reason is that the performance of deep graph-generative models (except SGAE) will significantly degrade when generating graphs with more than 1k nodes.
- In terms of graph simulation quality, GraphRNN can achieve the best overall performance and can mimic the structural distribution of observed graphs adequately. It is a good choice if there are sufficient resources, and the relevant costs are acceptable for users. For instance, GraphRNN can be used for applications (e.g., protein graphs in bioinformatics) where the sizes of observed graphs and simulated graphs are small (e.g., less than 1000 nodes in our experiment environment). Note that GraphRNN also demonstrates a good performance in model robustness and parameter-tuning difficulty.
- If the efficiency and the scalability are of high priority, DCSBM and BTER are recommended because both are very efficient and scalable and have a good overall performance on graph simulation quality among approaches where the deep neural networks are not applied. Note that although BTER outperforms DCSBM in terms of inference time, DCSBM can better preserve the community structures of observed graphs.
- If users look for a good balance between graph simulation quality and efficiency (scalability), SGAE, as proposed in this paper, is recommended. Compared to two RNN-based generators GRAN and BiGG, SGAE displays competitive performance in graph simulation quality, but outshines the others with better efficiency and scalability. Moreover, in our experiment settings, SGAE outperforms other autoencoder-based generators and GAN-based generators in both graph simulation quality and efficiency (scalability). As reported in Table 14, SGAE also achieves well in terms of model robustness, memory consumption, community preserving, and parameter tuning, compared to other deep neural networks-based approaches.
- When users are only interested in several specific properties of the observed graphs, other graph generators can be recommended. For instance, if users are only interested in the scale-free property of the graphs, the B-A model can comfortably fit this role. Similarly, the W-S model is recommended to simulate small-world graphs. When it is important to preserve the community structures of observed graphs, SBMGNN is a good choice.

6 Conclusion

Graph data simulation is fundamental in a wide range of applications such as social networks, e-commerce, and bioinformatics. In this paper, we fill important gaps in this line

Table 14 Overall performance evaluation of 20 representative general graph generators

Graph Generator	Output Quality	Training Time	Inference Time	Scalability	Space	Robustness	Community Preserving	Tuning Difficulty
E-R	★	–	★★★★½	★★★★½	★★★★★	–	–	★★★★★
W-S	★	–	★★★★★	★★★★½	★★★★★	–	–	★★★★
B-A	★	–	★★★★½	★★★★★	★★★★★	–	–	★★★★★
RTG	★	–	★★★½	★★★★½	★★★★★	–	–	★★★★
BTER	★★½	–	★★★★½	★★★★★	★★★★½	–	–	★★★★
SBM	★★	–	★★★½	★★★★½	★★★★★	–	★★	★★★★
DCSBM	★★½	–	★★★½	★★★★★	★★★★½	–	★★½	★★★★
R-MAT	★★	–	★★★★½	★★★★★	★★★★★	–	–	★★★★
Kronecker	★★	★★★★★	★★★★½	★★★★★	★★★★★	★★★	–	★★★★
MMSB	★★	★★★½	★★★½	★★★	★★	★★★	★★	★★★★
VGAE	★★★½	★★★★	★★★★	★★★½	★★★	★★★★	★★★½	★★★★
Graphite	★★★½	★★★★	★★★½	★★★½	★★★	★★★	★★★½	★★★½
SBMGNN	★★★½	★★★½	★★★	★★★½	★★★	★★★	★★★★	★★★½
GraphRNN	★★★★★	★	★	★	★	★★★★	–	★★★★
GRAN	★★★★½	★★	★★	★★★	★★	★★★	–	★★★
BiGG	★★★★½	★★½	★★★	★★★½	★★★★	★★★	–	★★★
ARVGA	★★★	★★★★	★★★	★★★½	★★★	★★★	★★★	★★
NetGAN	★★★½	★★★	★★½	★★★½	★★★	★★★★	★★★	★★★
CondGEN	★★★	★★	★★★½	★★	★★★★	★★	–	★★
SGAE	★★★★	★★★★½	★★★★	★★★★	★★★★	★★★★	★★★½	★★★★

The more amount of (★), the better the performance. Note that the symbol ½ is used to refine the ranks of the graph generators

of research by: (i) giving an overview of 20 representative general graph generators, including recently emerged deep learning-based approaches; (ii) conducting comprehensive experiments on 20 representative general graph generators and providing broad-spectrum recommendations for both researchers and practitioners; (iii) developing a new algorithm to achieve a good trade-off between graph simulation quality and efficiency; and (iv) implementing a user-friendly platform for researchers and practitioners such that they not only can easily apply a variety of existing general graph generators to their work but also immediately integrate their own general graph generators for comprehensive performance comparison and analytics.

Acknowledgements This work was supported by 2018YFB2100801, NSFC62102287, 19511101300. Ying Zhang is supported by FT170100128 and ARC DP210101393. Lu Qin is supported by ARC FT200100787. Xuemin Lin is supported by NSFC61232006, 2018YFB1003504, ARC DP200101338 and ARC DP180103096.

References

1. Airoldi, E.M., Blei, D.M., Fienberg, S.E., et al.: Mixed membership stochastic blockmodels[J]. JMLR **1**(9):1981–2014 (2008)
2. Akoglu, L., Faloutsos, C.: RTG: a recursive realistic graph generator using random typing[J]. Data Min. Knowl. Discov. **19**(2):194–209 (2009)
3. Albert, R., Barabási, A.-L.: Statistical mechanics of complex networks. *Reviews of modern physics*, page 47, 2002
4. Bacciu, D., Micheli, A., Podda, M.: Graph generation by sequential edge prediction. ESANN (2019)
5. Bacciu, D., Micheli, A., Podda, M.: Edge-based sequential graph generation with recurrent neural networks[J]. Neurocomputing **4**(16):177–189 (2020)
6. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gmark,: Schema-driven generation of graphs and queries. IEEE TKDE **856–869**,(2017)
7. Barrett, C. L., Beckman, R. J., Khan, M., Kumar, V. S. A., Marathe, M. V., Stretz, P. E. , Dutta, T., Lewis, B. L.: Generation and analysis of large synthetic social contact networks. In *WSC*, pages 1003–1014. IEEE, 2009
8. Batagelj, V., Brandes, U.: Efficient generation of large random networks. *Phys. Rev. E*, page 036113, 2005
9. Blondel, V. D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. *J. Stat. Mech.: Theory and Experiment*, page P10008, 2008
10. Bojchevski, A., Shchur, O., Zügner, D., Günnemann, S.: Netgan: Generating graphs via random walks. In: *ICML*, pp. 610–619, 2018
11. Bonifati, A., Holubová, I., Prat-Pérez, A., Sakr, S.: Graph generators: State of the art and open challenges. ACM Comput Surv **53**(2):1–30 (2020)
12. Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O.: Generative code modeling with graphs. ICLR. OpenReview.net (2019)
13. Bu, D., Zhao, Y., Cai, L., et al.: Topological structure analysis of the protein–protein interaction network in budding yeast[J]. Nucleic acids research **31**(9):2443–2450 (2003)
14. Cayley. On Monge's: "Mémoire sur la Théorie des Déblais et des Remblais". In: Proceedings of the London Mathematical Society, pp. 139–143 (1882)
15. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. In: *ICDM*, 2004
16. Chang, C., Lin, C.: LIBSVM: A library for support vector machines. ACM Trans. Intell. Syst. Technol. **27**(1–27), 27 (2011)
17. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. NeurIPS (2014)
18. Dai, H., Nazi, A., Li, Y., Dai, B., Schuurmans, D.: Scalable deep generative modeling for sparse graphs. In *ICML*, pages 2302–2312, (2020)
19. Dobson, D.P., Doig, J.A.: Distinguishing enzyme structures from non-enzymes without alignments. J. Mol. Biol. **771–783**,(2003)
20. Erdős, P., Rényi, A.: On random graphs i. publicationes mathematicae (debrecen). 1959
21. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014
22. Goyal, P., Ferrara, E.: Graph embedding techniques, applications, and performance: a survey. Knowledge-Based Syst. **151**, 78–94 (2018)
23. Grover, A., Zweig, A., Ermon, S.: Graphite: Iterative generative modeling of graphs. In *ICML*, pages 2434–2444, 2019
24. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A. C.: Improved training of wasserstein gans. In: *NeurIPS*, pages 5767–5777, 2017
25. Hagberg, A. A., Schult, D. A., Swart, P. J.: Exploring network structure, dynamics, and function using NetworkX. In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, 2008
26. Hochreiter, S., Schmidhuber, J.: Long short-term memory[J]. Neural comput. **9**(8):1735–1780 (1992)
27. Holland, P.W., Laskey, K.B., Leinhardt, S.: Stochastic block models: First steps[J]. Social networks **5**(2):109–137 (1983)
28. Jin, W., Barzilay, R., Jaakkola, T.: Junction tree variational autoencoder for molecular graph generation. In *ICML*, pages 2328–2337, 2018
29. Jin, W., Barzilay, R., Jaakkola, T.: Junction tree variational autoencoder for molecular graph generation, 2019
30. Joshi, A. K., Hitzler, P., Dong, G.: Linkgen: Multipurpose linked data generator. In *ISWC*, Lecture Notes in Computer Science, pages 113–121, (2016)
31. Karrer, B., Newman, M.E.J.: Stochastic blockmodels and community structure in networks[J]. Physical review E **83**(1):016–107 (2011)
32. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. ICLR (2014)
33. Kipf, T. N., Welling, M.: Semi-supervised classification with graph convolutional networks, 2016
34. Kipf, T.N., Welling, M.: Variational graph auto-encoders. NeurIPS (2016)
35. Kolda, T.G., Pinar, A., Plantenga, T.D., Seshadhri, C.: A scalable generative graph model with community structure. SIAM J. Sci, Comput (2014)
36. Kullback, S., Leibler, R. A.: On information and sufficiency[J]. Ann. Math. Statist. **22**(1):79–86 (1951)
37. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: An approach to modeling networks. JMLR **985–1042**,(2010)
38. Leskovec, J., Kleinberg, J. M., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: *SIGKDD*, pages 177–187. ACM, (2005)
39. Leskovec, J., Lang, K.J., Dasgupta, A., et al.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters[J]. Internet Math. **6**(1):29–123 (2009)

40. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R. S.: Gated graph sequence neural networks. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2016
41. Li, Y., Vinyals, O., Dyer, C., Pascanu, R., Battaglia, P.W.: Learning deep generative models of graphs. CoRR (2018)
42. Liao, R., Li, Y., Song, Y., Wang, S., Hamilton, W., Duvenaud, D. K., Urtasun, R., Zemel, R.: Efficient graph generation with graph recurrent attention networks. In *NeurIPS*, pages 4255–4265, 2019
43. Ma, T., Chen, J., Xiao, C.: Constrained generation of semantically valid graphs via regularizing variational autoencoders. In *NeurIPS*, page 7113–7124, 2018
44. Marcelli, A., Quer, S., Squillero, G.: The maximum common subgraph problem: A portfolio approach. CoRR (2019)
45. McCallum, A.K., Nigam, K., Rennie, J., et al.: Automating the construction of internet portals with machine learning[J]. Inf. Retr. **3**(2):127–163 (2003)
46. Mehta, N., Carin, L., Rai, P.: Stochastic blockmodels meet graph neural networks. In *ICML*, pages 4466–4474, 2019
47. Moreno, S., Neville, J., Kirshner, S.: Tied kronecker product graph models to capture variance in network populations. ACM TKDD (2018)
48. Neumann, M., Moreno, P., Antanas, L., et al.: Graph kernels for object category prediction in task-dependent robot grasping[C]//Online. In: Proceedings of the Eleventh Workshop on Mining and Learning with Graphs, p6 (2013)
49. Pan, S., Hu, R., Long, G., Jiang, J., Yao, L., Zhang, C.: Adversarially regularized graph autoencoder for graph embedding. In *IJCAI*, pages 2609–2615, 2018
50. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. In *SIGKDD*, pages 701–710. ACM, 2014
51. Podda, M., Bacciu, D., Micheli, A.: A deep generative model for fragment-based molecule generation[C]//International Conference on Artificial Intelligence and Statistics. PMLR 2240–2250 (2020)
52. Rezende, D. J., Mohamed, S., Wierstra, D.: Stochastic backpropagation and approximate inference in deep generative models. In *ICML*, JMLR Workshop and Conference Proceedings. JMLR.org, 2014
53. Robins, G., Pattison, P., Kalish, Y., Lusher, D.: An introduction to exponential random graph (p*) models for social networks. Social networks **29**(2):173–191 (2007)
54. Rozemberczki, B., Davies, R., Sarkar, R., Sutton, C.: Gemsec: Graph embedding with self clustering. In: *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019*, pages 65–72. ACM, 2019
55. Salha, G., Hennequin, R., Remy, J.-B., Moussallam, M., Vazirgiannis, M.: Fastgae: Scalable graph autoencoders with stochastic subgraph decoding. arXiv preprint arXiv:2002.01910, 2020
56. Sanfeliu, A., Fu, K.: A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Syst. Man Cybern. 353–362 (1983)
57. Sarkar, A., Mehta, N., Rai, P.: Graph representation learning via ladder gamma variational autoencoders[C]//Proceedings of the AAAI Conference on Artificial Intelligence **34**(04):5604–5611 (2003)
58. Sen, P., Namata, G., Bilgic, M., Getoor, L., Gallagher, B., Eliassi-Rad, T.: Collective classification in network data. *AI Mag.*, pages 93–106, 2008
59. Simonovsky, M., Komodakis, N.: Graphvae: Towards generation of small graphs using variational autoencoders. In *ICANN*, pages 412–422, 2018
60. Simonovsky, M., Komodakis, N.: In: In, I.C.A.N.N. (eds) Graphvae: Towards generation of small graphs using variational autoencoders, pp. 412–422. Springer (2018)
61. Stoyanovich, J., Howe, B., Jagadish, H.V.: Responsible data management. In: Proc. VLDB Endow. **3474–3488**,(2020)
62. Su, S., Hajimirsadeghi, H., Mori, G.: Graph generation with variational recurrent neural network. CoRR (2019)
63. Tai, K.S., Socher, R., Manning, C.D.: Improved semantic representations from tree-structured long short-term memory networks. In: ACL, pp. 1556–1566. The Association for Computer Linguistics (2015)
64. Teh, Y.W., Grür, D., Ghahramani, Z.: Stick-breaking construction for the indian buffet process. In: Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics, pages 556–563. PMLR, 2007
65. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. ICLR (2018)
66. Watts, D., Strogatz, S.: Collective dynamics of ”small-world” networks (see comments). Nature, pages pp .440–442 (1998)
67. Wu, L., Chen, Y., Shen, K., Guo, X., Gao, H., Li, S., Pei, J., Long, B.: Graph neural networks for natural language processing: A survey. CoRR (2021)
68. Wu, Z., Pan, S., Chen, F., Long, G., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE TNNLS **1–21**,(2020)
69. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE TNNLS **1–21**,(2020)
70. Xia, F., Sun, K., Yu, S., Aziz, A., Wan, L., Pan, S., Liu, H.: Graph learning: A survey. *CoRR*, abs/2105.00696, 2021
71. Xiao, H., Huang, M., Zhu, X.: Transg: A generative model for knowledge graph embedding. In: ACL, The Association for Computer Linguistics (2016)
72. Xie, S., Kirillov, A., Girshick, R. B., He, K.: Exploring randomly wired neural networks for image recognition. In: *ICCV*, pages 1284–1293. IEEE, 2019
73. Yang, C., Zhuang, P., Shi, W., Luu, A., Li, P.: Conditional structure generation through graph variational generative adversarial nets. NeurIPS (2019)
74. You, J., Leskovec, J., He, K., Xie, S.: Graph structure of neural networks. In *ICML*, pages 10881–10891, 2020
75. You, J., Liu, B., Ying, R., Pande, V., Leskovec, J.: In: NeurIPS, In., page, (eds.) Graph convolutional policy network for goal-directed molecular graph generation. Curran Associates Inc, pp. 6412–6422(2018)
76. You, J., Wu, H., Barrett, C. W., Ramanujan, R., Leskovec, J.: G2SAT: learning to generate SAT formulas. In *NeurIPS*, pages 10552–10563, 2019
77. You, J., Ying, R., Ren, X., Hamilton, W., Leskovec, J.: Graphrnn: Generating realistic graphs with deep auto-regressive models. In *ICML*, pages 5694–5703, 2018
78. Zhang, Z., Cui, P., Zhu, W.: Deep learning on graphs: a survey. IEEE TKDE, (2020)
79. Zhao, L., Akoglu, L.: Pairnorm: Tackling oversmoothing in gnns. ICLR (2020)
80. Zhou, D., Zheng, L., Han, J., He, J.: A data-driven graph generative model for temporal interaction networks. In *SIGKDD*, pages 401–411. ACM, 2020
81. Zhou, D., Zheng, L., Han, J., He, J.: A data-driven graph generative model for temporal interaction networks. In: *SIGKDD*, page 401–411, 2020